Integration de contraintes temps-reel au sein d'un processus de developpement incremental base sur la preuve (Livrable 2)

Dominique Cansell, Dominique Méry, Joris Rehm

Livrable 2
Intégration de contraintes temps-réel au sein d'un processus de développement incrémental basé sur la preuve.

4 août 2008

Projet RIMEL

ANR-06-SETI-015

Equipe MOSEL, LORIA, Université Henri Poincaré Nancy 1
ClearSy
LABRI, Université de Bordeaux & CNRS
http://rimel.loria.fr
Années 2007-2008-2009

Avertissement
Ce document a été rédigé par Joris Rehm ; il a été supervisé par Dominique Cansell.

Table des matières

Chapitre 1

Introduction

Ce livrable est un rapport sur l'intégration de contraintes temporelles dans les modèles *Event-B*. Nous entendons par contraintes temporelles, les propriétés que possède, ou que l'on veut imposer à, un système, par rapport à son déroulement dans le temps. On parle aussi de propriétés temps-réel. Les systèmes seront en particulier des systèmes distribués, en effet : mis à part une mesure de la performance, les propriétés temps-réel ne sont utiles que si l'on s'intéresse à un ensemble de systèmes fonctionnant de manière concurrente ou distribuée. Nous allons donc étudier des systèmes dynamiques, qu'ils soient logiciels ou matériels, les définir et démontrer des propriétés, en particulier temps-réel, sur ces modèles.

Nous nous attacherons à proposer une méthodologie incrémentale, les systèmes seront d'abord décrits sans aucun aspect temps-réel, puis ceux-ci seront ajoutés progressivement. Nous utiliserons la méthode formelle «B-évènementielle» (*Event-B method*). Il va de soi que les résultats sont transposables à une autre notation, à partir du moment où celle-ci est suffisamment proche et expressive.

L'étude des propriétés temps-réel n'est pas fondamentalement différente des autres domaines d'application des méthodes formelles. En particulier, il est rare qu'un utilisateur de méthodes formelles ait uniquement à étudier un problème temps-réel. En effet, la plupart du temps les problèmes avec des aspects temps-réel surviennent ponctuellement au sein d'un système plus grand. C'est pourquoi nous avons trouvé intéressant d'étudier comment mettre en œuvre les contraintes de temps au sein de la méthode B évènementielle, qui a déjà montré son efficacité dans un cadre général, mais qui ne dispose pas de concepts spécifiques pour le temps-réel.

Dans ce document nous commencerons, dans le chapitre 2, par introduire le concept d'agenda d'actions. Les systèmes que nous étudions sont dynamiques, ils possèdent un état courant et peuvent le modifier par des transitions entre les états. Les transitions représenteront les actions de notre système. Pour exprimer des contraintes temps réel, un des moyens est d'attacher à chaque action l'ensemble de ses futures occurrences. Nous appelons l'ensemble de ces ensembles un agenda, un agenda régule donc plusieurs actions dans le temps.

Cette définition de l'agenda sera ensuite encodée dans un patron avec la notation *Event-B*. Ce patron donne un moyen d'utiliser les concepts définis dans un développement prouvé, pour cela nous utilisons la relation de raffinement entre modèle, ce qui permet une approche pour appliquer les patrons. Nous donnerons des explications sur ce processus et un exemple d'utilisation.

Pour valider notre approche, nous verrons, dans le chapitre 3, une expérimentation sur une étude de cas, le protocole RCP (*Remote Contention Protocol*) qui est une partie du standard IEEE 1394 (*Firewire*). Dans cette étude de cas nous expérimentons notre patron d'agenda sur un système distribué de deux appareils qui communiquent à l'aide de signaux sur un câble. Nous montrons comment utiliser le raffinement pour introduire les concepts temps-réel et comment réitérer ce raffinement pour préciser ces même contraintes. Nous verrons que cette utilisation du raffinement permet de valider un développement prouvé par étapes et de séparer la preuve en plusieurs parties. En effet nous avons pu, dans un premier temps, traiter les problèmes non temps-réel, comme la communication et une première spécification de l'algorithme. Les propriétés temps-réel sont introduites dans le raffinement suivant et nous montrons que nous pouvons remplacer la spécification abstraite par des contraintes temps-réel concrètes qui expriment les propriétés de l'implémenta-

tion. Enfin nous montrons que notre approche nous a bien permis de vérifier que les contraintes temps-réel implémentent la spécification du système.

Cette méthodologie est appliquée à un autre concept : le compteur de durée d'un prédicat. Nous allons donc définir dans le chapitre 2 ce concept qui permet d'associer une valeur de durée à la véracité d'un prédicat dans le cadre d'un système dynamique. De même que pour l'agenda, nous présentons un patron qui donne un modèle de ce concept.

Et dans le chapitre 4, nous présentons une étude de cas qui expérimente ce patron sur un algorithme de communication asynchrone, l'algorithme de Simpson. Cet algorithme utilise des contraintes temps-réel pour éviter des accès concurents à une mémoire partagée. Nous verrons comment les durées associées à certains prédicats permettent d'exprimer ces contraintes et de vérifier formellement cet algorithme.

Chapitre 2

Modèle du temps

Nous allons définir dans ce chapitre les concepts qui nous permettrons d'étudier les propriétés temps-réel des systèmes. Nous commençons d'abord dans la section 2.1 par définir une structure appelée agenda qui permettra d'imposer le déroulement chronologique d'actions. Puis, dans cette même section, nous définissons un compteur de durée sur un prédicat, ce qui permettra de quantifier la durée de validité d'un prédicat.

Il s'agira ensuite de pouvoir utiliser ces concepts dans des modèles en Event-B. Pour cela nous procédons à leur encodage sous la forme de patrons de modèle Event-B. Par la suite ces modèles peuvent être appliqués et instanciés pour étudier un système. En particulier nous pourrons utiliser le raffinement pour appliquer ces patrons. La section 2.2 présente le patron de l'agenda d'actions en valeur absolue et la section 2.3 le patron du compteur de durée sur un prédicat.

2.1 Définitions

Nous allons étudier des systèmes dynamiques, c'est-à-dire que ces systèmes sont caractérisés par un état qui évolue selon certaines transitions. Par exemple un feux de signalisation tricolore est un système dont l'état est formé par la couleur des lampes allumées et dont les transitions précisent de quelle manière, et à quelles conditions, ces lampes sont sensées s'allumer ou s'éteindre.

Nous noterons $STATE$ l'ensemble des états du système étudié. Les transitions forment une relation sur l'ensemble $STATE$. Et nous attachons des étiquettes, de l'ensemble $TRANS$, aux transitions.

En pratique, l'ensemble $STATE$ des états est construit en choisissant des variables, et les transitions feront varier leurs valeurs. Soit X dénotant les variables du modèle, avec X_s la valeur de ces variables dans l'état s.

Nous allons considérer, dans toute la suite, une transition $tr \in TRANS$ entre l'état b et e $(b, e \in STATE^2)$:

$$b \xrightarrow{tr} e$$

où b (Begin) sera, dans chaque définition, l'état avant la transition tr et e (End) l'état après.

Nous souhaitons étudier le déroulement chronologique du système, pour cela nous associons la valeur $t_s \in \mathbb{T}$ à chaque état $s \in STATE$. La valeur t_s est l'instant au cours duquel le système étudié se trouve à l'état s.

Nous disposons d'un ensemble $\mathbb{T}$ dans lequel t prend sa valeur. Pour la vérification de nos modèles, il sera nécessaire de définir formellement $\mathbb{T}$; pour le moment nous allons supposer que nous disposons, au moins, d'une loi $+$ de composition interne et d'une relation $\leq$ d'ordre total. En pratique, dans les modèles formels, nous utiliserons l'ensemble des entiers naturels $\mathbb{N}$ qui est défini dans la méthode B.

Le système sera modélisé de telle manière que les transitions dénotant la progression temporelle du système auront pour label tic et n'auront pas d'effet sur les variables du modèle (si $tr = tic$ alors $t_b < t_e$ et $X_b = X_e$), le temps ne faisant que croître. Toutes les autres transitions sont libres de modifier les variables mais sont instantanées (si $tr \neq tic$ alors $t_b = t_e$).

2.1.1 Agenda d'actions en valeur absolue

Soit ACT un ensemble d'actions que l'on souhaite planifier dans le temps.

Définition 1 *Agenda d'actions at*

Soit $at_s(a)$ l'ensemble des occurrences futures de l'action $a \in ACT$ à chaque état $s \in STATE$ du système.

Notons F_s le prédicat dépendant d'un état $s \in STATE$ vérifiant : «$t_s \leq x$ avec x une occurence d'une action a ($x \in at_s(a)$ et $a \in ACT$)». Autrement dit : «toutes les occurrences des actions prévues sont dans le présent ou le futur».

Alors at est telle que :

1. Pour tous les états s : F_s (F est toujours vrai).
2. Pour une transition $tr \neq tic$, s'il existe une action $a \in ACT$ telle que $t_b \in at_b(a)$ alors le système doit effectuer l'action a dans la transition tr et nous avons at telle que $t_e \notin at_e(a)$ (rappelons que $t_b = t_e$). Notons que dans ce cas, puisque F est vrai pour tout état, nous avons : $t_b = min(at_b(a))$.

Cette structure est dite en valeur absolue car les valeurs temporelles ont pour référence le temps zéro.

2.1.2 Compteur de durée sur un prédicat

Soit $P(X)$ un prédicat sur X.

Définition 2 *Compteur de durée D*

Soit $D_s(P)$, pour un certain état $s \in STATE$, la dernière durée pendant laquelle le prédicat P est, ou était, vrai.

D est telle que :

1. Pour l'état initial i : $D_i(P) = 0$.
2. Pour une transition tr telle que $tr \neq tic$, si $\neg P(X_b)$ et $P(X_e)$ nous avons $D_e(P) = 0$.
3. Toujours avec $tr \neq tic$ et puisque les transitions différentes de tic sont instantanées, nous avons $D_e(P) = D_b(P)$ dans tous les autres cas que 2, c'est-à-dire : $P(X_b)$ et $P(X_e)$; $P(X_b)$ et $\neg P(X_e)$; $\neg P(X_b)$ et $\neg P(X_e)$.
4. Soit maintenant une transition $tr = tic$, rappelons que $t_b < t_e$, $X_b = X_e$ et notons $d = t_e - t_b$. Deux cas sont possibles si $P(X_b)$ alors $D_e(P) = D_b(P) + d$, sinon $\neg P(v)$ et $D_e(P) = D_b(P)$.

2.2 Patron de l'agenda d'actions en valeur absolue

Pour définir des spécifications temps-réel, nous allons ajouter des contraintes temps-réel à un modèle en B évènementiel. Ce travail a fait l'objet d'une publication [6]. Pour cela nous allons considérer l'ensemble ACT d'actions à ordonner dans le temps. Chaque action sera modélisée par au moins un évènement du modèle. Il est aussi possible qu'une action soit modélisée par plusieurs évènements au choix. Dans ce cas, l'action est considérée comme effectuée dès qu'un évènement a été exécuté. Les labels de transition $TRANS$ correspondent aux évènements.

Nous allons donc expliciter ici comment mettre en œuvre notre agenda dans un modèle B évènementiel. Nous appelons cette mise en œuvre un patron.

Afin de modéliser la progression du temps, nous utiliserons la variable now qui est incrémentée à chaque fois que le temps progresse. Comme son nom le suggère now est la date de l'instant présent, c'est-à-dire t_s si s est l'état courant.

Nous noterons at la fonction qui associe à une action $a \in ACT$ l'ensemble $at(a)$ de ses futures occurrences $at(a) \subseteq \mathbb{T}$, formellement $at \in ACT \rightarrow \mathbb{P}(\mathbb{T})$.

Ces valeurs se trouvent dans le futur, selon le prédicat F de la définition, les variables now et at doivent donc respecter l'invariant :

$$\forall a \cdot a \in ACT \wedge at(a) \neq \varnothing \Rightarrow now \leq \min(at(a))$$

FIG. 2.1 – Illustration de l'action de l'évènement *tic* : avant et après

FIG. 2.2 – Illustration de l'action de l'évènement *add* : avant et après

avec min le minimum d'un ensemble non vide.

Pour représenter correctement une modélisation temps-réel, il est nécessaire de pouvoir contraindre une action à se déclencher dans un certain délai. De plus, l'invariant ci-dessus suggère qu'il faut bloquer la progression du temps au moment auquel nous attendons le déclenchement. L'évènement ci-dessous nous permet d'assurer cela.

tic $\widehat{=}$
ANY
n_now
WHERE
grd1 : $n\_now \in \mathbb{T} \land now < n\_now$
grd2 : $\forall a \cdot a \in dom(at) \land at(a) \neq \varnothing \Rightarrow n\_now \leq min(at(a))$
THEN
act1 : $now := n\_now$
END

Sur la figure 2.1, on peut voir une illustration d'un déclenchement de l'évènement *tic*. Il s'agit de deux lignes temporelles, avec la position du temps courant *now* et trois occurrences x, y et z pour certaines actions non représentées. À gauche une représentation du système avant le déclenchement de *tic* et à droite après. On voit que *now* se déplace dans la zone délimitée par l'accolade, c'est-à-dire que la nouvelle valeur de *now* peut se trouver entre l'ancienne valeur de *now* et x.

Cela peut sembler contre-intuitif de bloquer le temps, et effectivement dans la réalité le temps s'écoule toujours. Mais il faut considérer que forcer une action à se produire à un moment donné, revient à interdire que le temps s'écoule au-delà de ce moment, si l'action n'a pas été réalisée.

Informellement, nous pouvons dire que ce premier évènement *tic* de notre modélisation provoque un saut non-déterministe du temps dans l'intervalle qui va de l'instant présent (exclu) jusqu'à la première occurence. Si rien n'a été prévu dans l'agenda, le temps est libre d'avancer.

Nous venons de voir comment la progression du temps est contrainte par les occurrences des actions. Pour ajouter dynamiquement de telles occurrences, on peut utiliser un évènement de la forme décrite ci-dessous.

add $\widehat{=}$
ANY
a
$ntime$
WHERE
grd1 : $a \in dom(at)$
grd2 : $now \leq ntime$
THEN
act1 : $at(a) := at(a) \cup \{ntime\}$
END

FIG. 2.3 – Illustration de l'action de l'évènement *use* : avant et après

La figure 2.2 représente l'ajout d'une nouvelle occurence *ntime*, pour une certaine action a (non représentée sur la figure).

Étant donnée une action a et une occurence *ntime* que l'on souhaite programmer, alors *add* ajoute la valeur *ntime* dans $at(a)$. Bien sûr, cela n'a de sens que si *ntime* est dans le futur. Cet évènement ne permet que d'ajouter des occurrences, sur le même modèle on peut concevoir un évènement plus général qui permet n'importe quelle modification (ajout, supression ou modification) à la condition de toujours travailler dans le futur.

Vient enfin le déclenchement des actions, cette activation est représentée par l'exécution d'un évènement de la forme suivante :

use $\widehat{=}$
ANY
a
WHERE
grd1 : $a \in dom(at)$
grd2 : $now \in at(a)$
THEN
act1 : $at(a) := at(a) \setminus \{now\}$
END

La figure 2.3 représente l'évènement *use* qui utilise une occurence, ici x, pour déclencher l'action qui y est associée. On voit à gauche l'occurence x qui a été supprimée.

Dans la garde de *add* nous trouvons, pour une certaine action a, la condition $now \in at(a)$ et nous savons par l'invariant que *now* est inférieur ou égal à tous les éléments de $at(a)$. Il s'ensuit que *now* est égal au plus petit élément de $at(a)$.

2.2.1 Utilisation

Le patron de la section précédente nous permet d'exprimer des contraintes de temps sur des évènements, c'est-à-dire d'exprimer le passage du temps et de contraindre le déclenchement d'actions à un moment précis. Ces occurrences peuvent être déterminées par d'autres évènements qui vont ainsi ajouter des contraintes temps-réel.

L'évènement *add* représente la création de contraintes dans le futur. L'évènement se déroulera dans un certain délai non nul dont la somme avec le temps courant est ajouté dans *at*. Pour utiliser cet évènement, il faudra l'adapter et le superposer avec l'évènement dont nous voulons qu'il ajoute des occurrences d'action. Par exemple, dans un système communiquant, l'envoi de message va contraindre l'évènement de réception de messages à se déclencher dans un certain delai.

On peut aussi considérer des modèles qui n'utiliseront pas *add* si les contraintes sont statiques et connues à l'avance. Par exemple pour représenter une horloge contrôlant un système qui s'active toutes les 10 unités de temps, l'initialisation est tel que $ran(at)$ contienne 0,10,20, ...

L'évènement *tic* fait avancer le temps courant représenté par la variable *now*. Le nouveau temps *n_now* est choisi strictement supérieur au temps courant, nous savons donc que la progression temporelle est assurée si l'évènement *tic* est déclenché. Il n'y a pas de pas d'incrémentation défini : tous les sauts et les valeurs possibles du temps sont potentiellement représentés. Ce n'est pas tant la valeur concrète de *now* qui est importante mais plutôt les interactions possibles entre les changements de valeurs de *now* et le contenu du co-domaine de *at*. Le fait d'autoriser tous les sauts possibles de temps permet aussi de raffiner l'évènement *tic* par une autre version de *tic* plus contrainte, avec plus d'actions à considérer. De la même manière, la valeur de *at* n'est pas concrètement connue mais doit être représentée par un invariant lors de l'adaptation du patron au

cas d'utilisation. Dans le cas où le nouveau temps *now* est égal à une occurence d'une action a alors cette action va pouvoir se déclencher et ainsi supprimer la contrainte formée par cette occurence, l'évolution du temps pourra ainsi progresser jusqu'à l'occurence suivante.

Nous proposons donc un patron qui, grâce à la représentation abstraite des contraintes de temps par la fonction *at*, permet d'être suffisamment adaptable. Néanmoins, certains cas d'utilisation peuvent requérir des adaptations et font l'objet de remarques supplémentaires ci-dessous.

Un évènement de type *add* n'est pas obligé d'ajouter un unique temps dans l'ensemble des temps d'activations. Il se peut très bien que l'on ait besoin d'ajouter plusieurs temps avec un seul évènement. Cela ne contredit en rien l'idée du patron. Pour poster plusieurs éléments, on peut effectuer plusieurs unions de variables, ou l'union d'un ensemble donné par extension, dans la substitution de *at* si le nombre de temps est constant et connu. Sinon il est toujours possible d'utiliser une substitution plus générale qui définit l'ensemble à ajouter par compréhension si le nombre de temps à rajouter est variable. Par rapport au processus d'utilisation du patron cela correspond au raffinement répété de l'évènement *add*.

Il n'est pas non plus contradictoire de considérer un évènement qui raffine à la fois « add » et « use », autrement dit que cet évènement se déclenche suivant une contrainte de temps et poste d'autres contraintes de temps pour lui-même ou pour être consommé par des évènements distincts.

Un cas plus délicat se pose quand on veut que plusieurs actions se déclenchent dans le même instant. C'est-à-dire lorsque que l'on étudie un système distribué ou parallèle. Pour toute action a appartenant à ACT, $at(a)$ est un ensemble, nous ne pouvons donc pas exprimer plusieurs activations d'une action a dans le même instant. Par contre il est possible que les différentes actions de ACT se déclenchent en même temps. Si l'on considère un ensemble P d'entités pouvant évoluer en parallèle, alors il faut prendre garde à ce que chaque action a d'un élément p de P soit distinct, nous pouvons, par exemple, les noter $a(p)$.

2.2.2 Exemple d'utilisation : un interrupteur minuteur de lampe

Pour illustrer le patron, nous pouvons prendre en tant qu'exemple un système composé d'une lampe et d'un interrupteur qui coupe automatiquement la lampe au bout d'un certain temps. Nous avons donc ici une seule action : $ACT = \{off\}$. Un utilisateur peut actionner l'interrupteur et nous observerons alors l'évènement *switch_on*. Un certain temps après nous observerons l'évènement *switch_off* qui éteint la lampe. Ces évènements se trouvent dans la figure 2.4.

En plus des variables *at* et *now* du patron, le modèle comprend la variable *light_on* qui représente l'état de la lampe. L'évènement *switch_on* fait passer la variable *light_on* à vraie (dans l'action *act*1) et ajoute une occurence d dans 10 ± 1 unité de temps pour l'action *off* (dans l'action *act*2). Cet évènement raffine *add* du patron. L'occurence d ajoutée est choisie dans l'intervalle $9..11$ pour représenter la possible imprécision de la minuterie.

L'évènement *switch_off* se déclenche donc 10 unités (plus ou moins une) de temps après le déclenchement de *switch_on*. Et cet évènement raffine l'évènement *use* du patron.

Finalement, on peut voir l'évènement *tic*, qui est identique à celui du patron.

Les invariants de ce modèle sont :

$$\text{inv1} : at \in act \rightarrow \mathbb{P}(\mathbb{N})$$

$$\text{inv2} : light\_on \in BOOL$$

$$\text{inv3} : light\_on = FALSE \Leftrightarrow at(off) = \varnothing$$

$$\text{inv4} : card(at(off)) \leq 1$$

L'invariant *inv*1 provient du patron ; *inv*2 donne le type de *light_on* qui est l'état de la lampe, $BOOL$ est l'ensemble des booléen ; *inv*3 nous dit que l'agenda pour l'action *off* est vide si et seulement si la lampe est éteinte ; et avec *inv*4 nous savons qu'il y a au plus une valeur dans l'agenda de *off*.

init $\widehat{=}$
  BEGIN
    act1 : now := 0
    act2 : at := { off $\mapsto \varnothing$ }
    act3 : light_on := FALSE
  END

switch_on $\widehat{=}$
  ANY
    d
  WHERE
    grd1 : d$\in$ 9.. 11
    grd2 : light_on=FALSE
  WHERE
    act1 : light_on := TRUE
    act2 : $at(off) := at(off) \cup \{now + d\}$
  END

switch_off $\widehat{=}$
  WHEN
    grd1 : now $\in$ at(off)
  THEN
    act1 : $at(off) := at(off) \setminus \{now\}$
    act2 : light_on := FALSE
  END

tic $\widehat{=}$
  ANY
    n_now
  WHERE
    grd1 : now $<$ n_now
    grd2 :$\forall$ a$\cdot$a$\in$dom(at) $\wedge$ at(a)$\neq \varnothing \Rightarrow$n_now$\leq$min(at(a))
  THEN
    act1 : now := n_now
  END

FIG. 2.4 – Évènement de l'exemple du minuteur

2.2.3 Exemple d'utilisation : passage d'un message

En tant qu'exemple d'utilisation du patron nous proposons le modèle d'un système composé de deux agents communiquants et d'un unique message passé entre les deux. Le but est d'utiliser un raffinement avec le temps pour exprimer le fait qu'un nœud sait que l'autre a reçu son message car il s'est passé suffisamment de temps pour que le message transite sur le média.

Spécification du problème

Comme toujours en B, nous commençons par mettre en place le contexte et le problème dans un modèle. Le premier modèle p0, cf figure 2.5 page 13, comporte un ensemble explicite de nœuds N composés des éléments a et b. Les variables du modèle comportent :

MODEL
$p0$
SETS
$N = \{a, b\}$
VARIABLES
A, S
$AB,$
B
INVARIANT
$A \subseteq \{a\} \quad \wedge$
$B \subseteq \{b\} \quad \wedge$
$AB \subseteq \{a\} \quad \wedge$
$S \subseteq \{a\} \quad \wedge$
$(A \neq \varnothing \implies AB \neq \varnothing)$
INITIALISATION
$A := \varnothing \quad ||$
$B := \varnothing \quad ||$
$AB := \varnothing \quad ||$
$S := \varnothing$

EVENTS
sendA $\widehat{=}$
when
$A = \varnothing$
then
$A := \{a\} \;||\; AB := \{a\}$
end
recB $\widehat{=}$
when
$AB = \{a\}$
then
$B := \{b\} \;||\; AB := \varnothing$
end
quA $\widehat{=}$
when
$B = \{b\}$
then
$S := \{a\}$
end
end

FIG. 2.5 – p0

On peut voir dans la figure 2.6 la suite des états du systèmes.

- La variable A qui est vide quand a n'a pas encore envoyé son message et égale à $\{a\}$ sinon, autrement dit c'est un booléen local à a qui se « souvient » si le nœud a déjà envoyé un message ou pas. Nous avons aussi la variable S qui passe de $\varnothing$ à $\{a\}$ pour signifier que le nœud a considère le message comme reçu.
- Un canal de transmission AB sous la forme d'un ensemble pouvant contenir l'auteur du message envoyé.
- La variable B est aussi utilisée comme un booléen local à b et permet de noter qu'un message a été reçu.

L'initialisation des variables est simple : tout est mis à vide. L'invariant exprime le fait que seul a peut envoyer un message et qu'il ne peut pas y avoir de message dans le canal de transmission si a n'a pas fait d'émission. Les autres clauses de l'invariant sont du typage exprimant ce que nous avons déjà dit dans la liste ci-dessus. Maintenant regardons comment le système va évoluer avec ses évènements :

- *sendA* peut se déclencher tant que a n'a pas déjà envoyé de message, il note que l'envoi a été fait avec la variable A et place le message dans le canal AB.
- *recB* se déclenche quand un message est en transit, il retire le message du canal AB et enregistre qu'il a été reçu par b.
- *quA* représente la prise d'information de a sur l'état de b. Comme il s'agit d'un modèle abstrait spécifiant le problème à résoudre, nous interrogeons directement b pour savoir si $B = \{b\}$ et

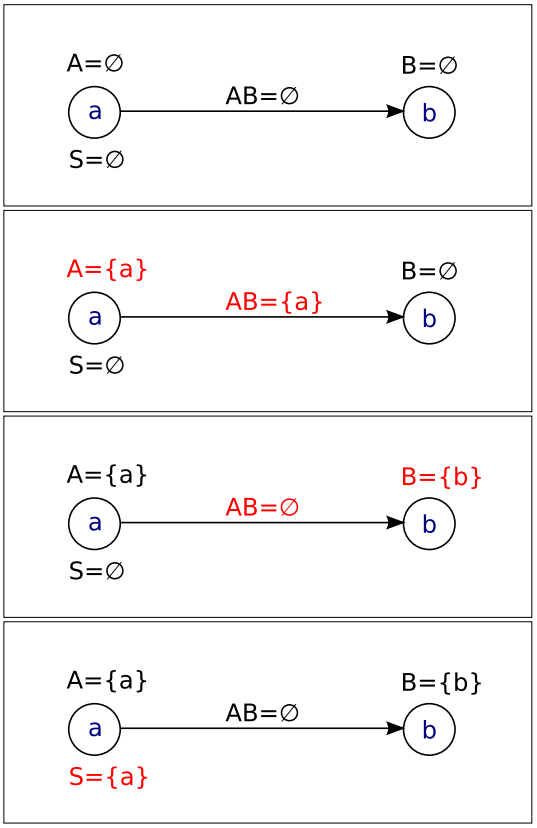

FIG. 2.6 – Séquence des états de l'exemple d'échange des messages

dans ce cas l'évènement est observé, S devient égale à $\{a\}$. Ce qui signifie que a sait que b a effectué la réception.

Comme nous sommes dans le cadre d'un système distribué sur des équipements distincts, il faut voir l'évènement *sendA* comme local au nœud a et *recB* local au nœud b. L'évènement *quA* est à ce propos spécial car il utilise des informations qui sont local à a et à b, ce qui n'est pas un comportement concret. Le but du raffinement suivant est justement de localiser ce comportement en utilisant des informations temporelles.

Application du patron

Nous avons donc écrit un deuxième modèle p1 figure 2.7 page 15 qui raffine le précédent. Nous allons rajouter des contraintes temporelles pour « ancrer » nos évènements dans le temps et pouvoir exprimer des propriétés nouvelles grâce à ces contraintes. Deux constantes apparaissent : *prop* et *st*, ce sont toutes les deux des entiers non nuls, elles expriment respectivement le temps qu'il faut à un message pour se propager et le temps que va attendre l'expéditeur après avoir effectué l'envoi. Nous avons évidemment $prop < stm$ sinon on ne pourra jamais introduire le comportement attendu. D'ailleurs si cette propriété n'était pas présente, elle aurait été imposée par la preuve.

Nous avons donc utilisé notre patron. Par souci de simplicité nous avons raffiné la fonction *at* en un ensemble *at*.

Le modèle comporte aussi deux variables entières supplémentaires :

- *stm* pour « Send Time Message » qui permet de stocker l'heure d'expédition du message
- *slp* pour « sleep » qui permet d'exprimer jusqu'à quel moment le nœud a dort après son émission.

Avant de voir quels invariants on peut maintenant écrire, passons en revue les changements faits aux évènements et l'adaptation du patron d'agenda. On peut déjà remarquer que l'évènement *tic* permettant de faire évoluer le temps a été réutilisé tel quel. L'évènement *add* se superpose sur *sendA*. C'est-à-dire que la nouvelle version de *add* est la fusion de l'ancienne version et de

REFINEMENT
$p1$
REFINES
$p0$
CONSTANTS
$prop$,
st
PROPERTIES
$prop \in \mathbb{N}^* \wedge$
$st \in \mathbb{N}^* \wedge$
$prop < st$
VARIABLES
A, AB, B, S
now,
stm,
slp,
at
INVARIANT
(cf figure 2.8)
THEOREMS
$(A \neq \varnothing \wedge now = slp \implies B = \{b\})$
INITIALISATION
$A, B, AB, S := \varnothing, \varnothing, \varnothing, \varnothing ||$
$now := 0 ||$
$at := \varnothing ||$
$stm := 0 ||$
$slp := 0$

EVENTS
$sendA \ \widehat{=}$
when
$A = \varnothing$
then
$A := \{a\} || AB := \{a\} ||$
$at := at \cup \{now + prop\}$
$\cup \{now + st\} ||$
$stm := now ||$
$slp := now + st$
end
$recB \ \widehat{=}$
when
$AB = \{a\} \wedge$
$now = stm + prop$
then
$B := \{b\} ||$
$AB := \varnothing ||$
$at := at - \{now\}$
end
$quA \ \widehat{=}$
when
$A \neq \varnothing \wedge$
$now = slp$
then
$S := \{a\} ||$
$at := at - \{now\}$
end
$tic\_tac \ \widehat{=}$
any tm **where**
$tm \in \mathbb{N}$
$\wedge now < tm$
$\wedge (at \neq \varnothing \implies tm \leq min(at))$
then
$now := tm$
end

FIG. 2.7 – p1

$$
\begin{array}{l}
now \in \mathbb{N} \\
\wedge stm \in \mathbb{N} \\
\wedge slp \in \mathbb{N} \\
\wedge at \subseteq \mathbb{N} \\
\wedge (A \neq \varnothing \implies stm + prop < slp) \\
\wedge (A \neq \varnothing \wedge now \geq stm + prop \wedge stm + prop \notin at \implies B = \{b\}) \\
\wedge (at \neq \varnothing \implies now \leq min(at)) \\
\wedge at \subseteq \{stm + prop, slp\} \\
\wedge (A = \varnothing \implies at = \varnothing) \\
\wedge (A \neq \varnothing \wedge at = \varnothing \implies now \geq slp) \\
\wedge (A \neq \varnothing \wedge at \neq \varnothing \implies slp : at) \\
\wedge (A \neq \varnothing \wedge at = \{slp\} \implies now \geq stm + prop)
\end{array}
$$

FIG. 2.8 – Invariant de p1

l'évènement du patron instancié avec les paramètres spécifiques à cette étude de cas. Quant à *use* il se superpose sur *recB* et *quA*. Expliquons maintenant les paramètres spécifiques à cette étude de cas :

- *sendA* va ajouter deux temps dans l'agenda *at* : il s'agit de $now + prop$ et $now + st$, avec *now* le temps courant. Comme le délai entre le temps courant et les occurrences futures est constant, le recours à la variable *ntime* du patron n'a pas été nécessaire.
- *recB* va quant à lui progresser par rapport à l'occurence précédemment ajoutée sur le temps de propagation des messages. Le temps courant doit respecter $now = stm + prop$ pour que l'on puisse observer l'évènement. Il s'agit donc de la modélisation de l'arrivée du message, ce qui appartient en partie à la modélisation de l'environnement, tout développement B devant respecter l'hypothèse du système clos. Conformément au patron, le temps courant est retiré de l'ensemble *at*. Comme il s'agit d'un raffinement, le comportement du modèle parent est toujours vrai, le message est donc retiré du canal et *b* change son état en conséquence.
- *quA* utilise l'autre contrainte qui « réveille » l'émetteur après que l'on est sûr que le message est arrivé. L'évènement se déclenche quand $now = slp$ avec $slp = now + st$ et comme d'habitude on supprime ce temps de *at*. Ce qui est intéressant est que l'on a réussi à raffiner la garde qui était $B = \{b\}$ en $A \neq \varnothing \wedge now = slp$. Nous avons donc localisé les informations nécessaires au déclenchement sur le nœud *a*. Pour cela, il faut prouver l'obligation de preuve de raffinement à l'aide des invariants que l'on a découverts lors du développement.

Nous ne détaillerons pas tout l'invariant, mais seulement trois clauses parmi les plus intéressantes ainsi qu'un théorème.

- $(A \neq \varnothing \wedge stm + prop \notin at \wedge now \geq stm + prop \Rightarrow B = \{b\})$: C'est cette partie de l'invariant qui sert finalement à localiser *quA*, il ne donne pas directement la formule qui permet de prouver le raffinement mais participe de manière centrale à sa preuve. En effet, à condition que le message soit parti, que le temps soit au moins à $stm + prop$, c'est-à-dire le moment où le message arrive, et que l'occurence $stmp + prop$ ait été traitée alors on est sûr que *b* a reçu le message ($B = \{b\}$).
- $(A \neq \varnothing \wedge at = \{slp\} \Rightarrow now \geq stm + prop$: Ici nous pouvons voir que si l'ensemble des occurrences est réduit à *slp* alors le temps courant a dépassé le moment où l'on reçoit le message.
- $(A \neq \varnothing \wedge at = \varnothing \Rightarrow now \geq slp$: Ce prédicat est intéressant, si le message a déjà été envoyé ($A \neq \varnothing$) et si il n'y a plus d'occurrences à traiter ($at = \varnothing$), autrement dit une fois que tous les évènements ont été observés. Dans ce cas là, on peut affirmer que le temps courant a dépassé le moment où *a* se réveille.
- $(A \neq \varnothing \wedge now = slp \implies B = \{b\})$ Enfin ce théorème déduit à partir de l'invariant donne l'information nécessaire au raffinement de *quA*.

Au final, ce prototype nous a permis de valider nos choix de modélisation et de nous conforter dans la possibilité de manipuler des contraintes temps-réel ainsi que de déduire des propriétés avec des

arguments temps-réel. On peut aussi voir une première utilisation de notre patron et voir comment il donne un cadre pour faciliter la résolution d'un problème d'un genre particulier. En effet, le cadre fournit par le patron permet de nous préoccuper seulement des problèmes spécifiques au système étudié et de gagner du temps pour la modélisation en tirant partie du savoir déjà acquis par l'expérience.

2.3 Patron du compteur de durée

Pour encoder le compteur D de la définition 2 en tant que variable d'un modèle nous avons aussi besoin d'encoder le prédicat P en tant que fonction booléenne, avec S un ensemble d'éléments représentant les prédicats. En effet, les prédicats ne peuvent être utilisés tels quels dans un modèle B, nous les transformons donc en état booléen. Ceci n'est présent que dans le patron, car dans le modèle final il conviendra de remplacer les prédicats par les véritables propriétés que l'on souhaite étudier.

Ci-dessous nous trouvons les variables, l'invariant et l'initialisatin du patron :

VARIABLES

D

P

INVARIANTS

$inv1: \ D \in S \rightarrow \mathbb{T}$

$inv2: \ P \in S \rightarrow BOOL$

EVENTS

Initialisation

begin

$act1: \ D := \{x \mapsto 0 | x \in S\}$

$act2: \ P :\in S \rightarrow BOOL$

end

La variable D est une fonction ($D \in S \rightarrow \mathbb{T}$), qui à chaque élément étudié retourne la durée mesurée. La variable P est l'encodage des prédicats en tant que fonction booléenne ($P \in S \rightarrow BOOL$). L'initialisation applique la valeur zéro pour toutes les durées et place indifféremment la valeur vrai ou faux à tous les prédicats. Les éléments de S serviront d'identifiants représentants les prédicats étudiés.

Nous pouvons maintenant définir l'évènement *bt* (Begin True) qui devra être utilisé à chaque fois qu'un prédicat étudié devient vrai pour la première fois :

Event bt $\widehat{=}$

any

x

where

$grd1: \ x \in S$

$grd2: \ P(x) = FALSE$

then

$act1: \ D(x) := 0$

$act2: \ P(x) := TRUE$

end

Nous avons donc, pour un identifiant x de prédicat, la valeur $P(X)$ qui passe de vrai à faux, et dans ce cas nous devons remettre à zéro le compteur de durée associé.

L'évènement suivant représent la progression du temps :

Event tic $\widehat{=}$

any

s

where

$grd1 : \ s > 0$

then

$act1 : \ D : |\forall x \cdot x \in S \Rightarrow ($
$(P(x) = TRUE \Rightarrow D'(x) = D(x) + s) \wedge$
$(P(x) = FALSE \Rightarrow D'(x) = D(x))$
$)$

end

END

Dans *tic* le temps progresse de s unités et si une valeur $P(x)$ est vrai alors il faut incrémenter de s le compteur $D(x)$ correspondant.

Dans l'utilisation de ce patron, il faut prendre garde à insérer l'évènement bt (Begin True) à chaque fois que le prédicat P devient vrai. Pour cela, il est possible de prouver dans un premier temps un modèle qui établit quand ce prédicat devient vrai. Et ensuite on raffine ce modèle pour superposer bt dans les bons évènements.

De même, pour utiliser la forme de *tic* il faut recourir à un modèle abstrait que l'on raffine. En effet il n'est pas possible de quantifier sur un prédicat (logique de second ordre) dans la notation de la méthode B. Il convient donc d'encoder ce prédicat dans une variable de type booléen comme il est montré dans le patron. De cette manière, nous pouvons effectivement utiliser le concept de compteur de durée sur un prédicat dans un modèle B.

2.4 Conclusion

Nous avons maintenant définit deux concepts pour étudier les systèmes temps-réel et dans la suite du document nous allons les expérimenter dans des études de cas.

Les deux concepts que nous avons définit sont duaux, en effet, l'agenda d'action permet de contraindre le déroulement d'un système dans le temps tandis que le compteur de durée permet de mesurer le comportement d'un système dans le temps.

Ces concepts ont d'abord été définis par les propriétés qu'ils doivent vérifier sur un système de transition, puis nous avons montré comment obtenir un modèle $Event - B$ qui servira de patron de raffinement pour utiliser ces concepts, en pratique, dans un développement prouvé.

Chapitre 3

Étude de cas : *Remote Contention Protocol*

3.1 Introduction

Nous présentons ici un modèle du protocole *IEEE 1394 Root Contention Protocol* (RCP) avec une correction partielle et une preuve des propriétés temps-réel de RCP. Ce travail a fait l'objet d'une publication [8].

Nous allons utiliser notre patron de contraintes temporelles dit d'agenda d'actions en valeurs absolues, avec la méthode B évènementielle, en tant que patron de raffinement pour introduire de manière incrémentale les propriétés temps-réel de RCP.

Le protocole standard FireWire (IEEE 1394) est une norme créée pour gérer la communication de périphériques informatiques. Ce protocole permet de connecter tout un ensemble de matériels entre eux. Les nœuds du réseau ainsi constitué peuvent être directement reliés à plusieurs autres à la manière d'un switch. Le réseau peut donc avoir une topologie libre selon le branchement que réalise l'utilisateur, la seule limitation topologique est qu'il est interdit de former des boucles. À noter qu'un ordinateur de bureau est considéré comme un périphérique comme les autres. Le protocole est utilisé dans de nombreux périphériques et ordinateurs du commerce. L'utilisateur de ce système n'a pas à réaliser la configuration des liaisons, les périphériques doivent être capables de se configurer de manière autonome. Pour cela, une phase d'initialisation du réseau (appelée *Bus Reset*) est lancée dès qu'un ou plusieurs périphériques sont branchés ou débranchés, cette phase d'initialisation a pour but "d'élire" un leader unique qui servira à réguler le réseau. Le sujet qui nous intéresse ici est cette phase d'initialisation. Algorithmiquement, cela correspond à trouver un arbre enraciné, la racine étant le leader, au sein d'un graphe non-orienté acyclique.

L'élection du leader, *Tree Indentify Protocol* dans la norme, se réalise par soumission des nœuds, chaque nœud "feuille" dans le graphe, c'est-à-dire relié à un seul autre périphérique, décide qu'il ne sera pas le leader et transmet sa décision à son unique voisin. Ensuite on considère qu'un nœud soumis ne fait plus partie du graphe et la procédure se poursuit en vague jusqu'à ce qu'il ne reste plus qu'un nœud. Un développement prouvé avec la méthode B évènementielle de cette partie d'initialisation de la norme a déjà été réalisée [2, 3] mais celle-ci laisse de côté les aspects temps-réel. On peut trouver une comparaison des travaux sur ce protocole dans [10].

En effet, il se peut que les messages de soumission des deux derniers nœuds se croisent dans la liaison auquel cas l'élection ne peut pas se faire. Le protocole *Remote Contention Protocol* (RCP) de détection et de résolution est alors appliquée pour ce cas dit de contention. Ce protocole fait appel à des mécanismes temps-réel, que nous allons modéliser.

3.2 Contention

Le problème de la contention est lié au décalage existant entre le moment où le message de soumission est parti et le moment où il arrive.

Si la contention est détectée dans le système, elle ne peut que concerner que deux nœuds. Ceci est montré dans [2, 3]

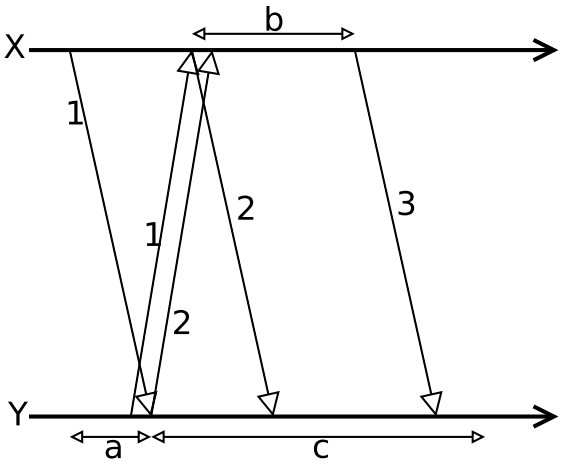

FIG. 3.1 – Échanges de message entre X et Y pendant la contention. On peut observer en *1* la propagation du signal de soumission et en *2* son retrait. En a la contention peut apparaître. Le temps d'attente court est b, le long est c. Enfin en *3* le message réglant la contention.

Pour que cela soit possible, les deux périphériques tirent un temps d'attente au hasard parmi deux possibles : un court et un long. Les messages ne peuvent être envoyés tant que les périphériques sont en attente. Si les deux nœuds choisissent le même temps, qu'il soit court ou long, la même situation se reproduit, le problème est détecté et une nouvelle itération est déclenchée. Il faut que les deux temps choisis soient différents pour que l'algorithme aboutisse et discrimine un leader. Le leader sera celui qui a tiré le temps le plus long, et pour cela il faut que ce temps soit suffisamment long pour que l'autre machine puisse venir à bout de son temps court, envoyer son message et que le message arrive à l'autre périphérique. Quand l'autre machine sort de son temps d'attente long, elle ne peut que constater la soumission et est ainsi forcée à devenir leader. Vous pouvez vous référer à la figure 3.1 page 20. Les section suivantes vont modéliser ces mécanismes formellement.

Nous n'avons pas besoin dans notre étude de prendre en compte l'équiprobabilité dans le tirage des délais, nous remplaçons donc le tirage aléatoire par le non-déterminisme. Les erreurs de transmissions ne sont pas non plus modélisées. Notons que les périphériques, à ce stade du protocole, communiquent par signaux continus et non pas par paquets.

3.3 Premier modèle : spécification de l'élection

Ce premier modèle est la spécification la plus abstraite de notre système. Le comportement général est de choisir (élire) un appareil dans l'ensemble N des appareils $N = \{a, b\}$. L'unique variable *leader* est un sous-ensemble de N et contient les appareils élus. L'invariant crucial du système est que l'on ne peut pas choisir deux appareils mais seulement un seul : *leader* est égale à $\{a\}$ ou $\{b\}$ ou $\varnothing$.

Bien sur, à l'initialisation la variable *leader* est égale à $\varnothing$.

Finalement, les transitions du systèmes sont données par l'évènement *accept*, montré dans la figure 3.2. Cet évènement se produit quand aucun appareil n'est élu ; *accept* en choisit un. Après cela, la garde est toujours fausse, d'autres transitions ne sont donc pas possibles. Tous les modèles que nous allons introduire devront raffiner ce comportement.

accept $\widehat{=}$
  ANY x WHERE
    x ∈ N ∧
    $leader = \varnothing$
  THEN
    leader := {x}
  END

FIG. 3.2 – Évenement accept

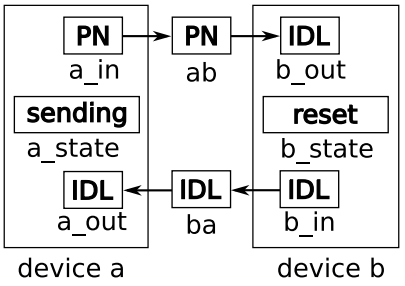

FIG. 3.3 – Appareils et canaux

3.4 Deuxième modèle : canaux et appareils

Nous allons maintenant introduire, par un raffinement, l'état local des deux appareils a et b et deux canaux de communications entre ces appareils. La plupart des comportements que l'on veut étudier sur ce système pourront déjà être introduits dans ce modèle, mais nous n'allons pas déjà quantifier les aspects temporels. Les communications sont, bien sûr, asynchrones.En d'autres termes, un signal partant de a n'arrive pas instantanément à b, d'autres évènements peuvent survenir avant.

Le système peut progresser de deux manières : si seulement un signal est envoyé, un leader va être élu directement ; si deux signaux se croisent, la contention apparaît. Dans cette situation, discriminer un appareil est impossible, les appareils vont commencer par supprimer leurs signaux.

Soit deux appareils distincts x et y, les comportements possibles sont : soit x émet un signal et y l'accepte ; soit x émet un signal et y aussi, dans ce dernier cas le processus reprend.

$SIGNALS$ sera l'ensemble des signaux dont disposent les appareils pour communiquer, $SIGNALS = \{IDL, PN\}$. IDL est le signal au repos, le signal PN (*Parent Notify*) exprime que l'émetteur du signal élit le récepteur du signal.

Nous avons 4 différent états dans l'ensemble $STATES$. Chaque appareil possédera un état à un moment donné : *reset* au démarrage ; *sending* quand l'appareil envoie le signal PN ; *sleeping* quand il attend avant de ré-émettre ; *accepting* quand il est devenu le leader.

Dans le processus de raffinement, la variable *leader* disparaît et nous ajoutons 9 nouvelles variables : a_state et b_state pour les deux appareils et trois variables pour chaque canal de communication a_in, ab, b_out and b_in, ba, a_out. Le nom des ces variables est choisi par rapport aux canaux et non par rapport aux variables. a_in est donc la variable représentant l'entrée du canal de a vers b. Nous avons besoin de trois variables par canal car cela permet de représenter la communication asynchrone en considérant qu'il n'y aura pas plus de deux changements de signaux au même moment. Ces trois variables se comportent comme une file "premier arrivé, premier sorti". Finalement, la variable *case* sera utile pour discriminer une condition dans l'invariant mais ne jouera pas de rôle dans le comportement du système (elle n'apparaît dans aucune garde). La représentation graphique de la figure 3.3 montre, en tant qu'exemple, l'état du système à la première émission d'un signal PN.

3.4.1 Invariant

Comme toujours, nous commençons par l'invariant de typage :

$a\_in, b\_in, a\_out, b\_out, ab, ba \in SIGNALS \times SIGNALS \times SIGNALS \times SIGNALS \times SIGNALS \times SIGNALS$

$a\_state, b\_state \in STATES \times STATES$

$case \in$ BOOL∧

Comme nous avons réalisé un raffinement de données en supprimant la variable *leader* du premier modèle, il faut donner un invariant de "collage" qui exprime les liens entre cette variable et les nouvelles. Nous avons en particulier :

$a\_state = accepting \Leftrightarrow leader = \{a\}$

et

$$(PN \in \{a\_in, ab, b\_out\} \wedge PN \in \{b\_in, ba, a\_out\}) \Rightarrow leader = \varnothing$$

Notons que le booléen *case* est vrai si et seulement si un des appareils est dans l'état *sending* et que l'autre dans l'état *sleeping*. L'appareil actuellement *sending* était juste auparavant *sleeping*. Autrement dit, *case* discrimine l'état ou l'un des appareils a déjà essayé de ré-émettre et pas l'autre, ce que nous pouvons voir dans les évènements.

3.4.2 Évènements

Nous avons ici quatre genre d'évènements : *send*, *pass*, *accept* et *sleep*. Le système est totalement symétrique entre l'appareil a et l'appareil b. Nous allons donc seulement décrire la partie concernant de a.

init $\widehat{=}$ BEGIN
  $a\_in, b\_in, a\_out, b\_out, ab, ba :=$
    $IDL, IDL, IDL, IDL, IDL, IDL \|$
  $a\_state, b\_state := reset, reset \|$
  $case :=$ FALSE
END;
a_send $\widehat{=}$ WHEN
  $a\_state = reset \wedge$
  $a\_out = IDL$
THEN
  $a\_state := sending \|$
  $a\_in := PN \|$
  $ab := PN$
END;
b_send $\widehat{=}$ ...
ab_pass_out $\widehat{=}$ WHEN
  $ab \neq b\_out \wedge$
  $(b\_state \neq sending \vee b\_out \neq PN)$
THEN
  $b\_out := ab \|$
  $ab := a\_in$
END;
ba_pass_out $\widehat{=}$ ...
pass_out $\widehat{=}$ WHEN
  $ab \neq b\_out \wedge$
  $ba \neq a\_out$
THEN
  $b\_out := ab \|$
  $ab := a\_in \|$
  $a\_out := ba \|$
  $ba := b\_in$
END;
a_accept $\widehat{=}$ REFINES accept WHEN
  $a\_state = reset \wedge$
  $a\_out = PN$
THEN
  $a\_state := accepting$
END;
b_accept $\widehat{=}$ ...
a_sleep $\widehat{=}$ ANY new_ab WHERE
  $a\_state = sending \wedge$
  $a\_out = PN \wedge$
  $new\_ab \in SIGNALS \wedge$
  $(ab = b\_out \Rightarrow new\_ab = IDL) \wedge$
  $(ab \neq b\_out \Rightarrow new\_ab = PN)$
THEN
  $a\_state := sleeping \|$
  $a\_in := IDL \|$
  $ab := new\_ab$
END;
b_sleep $\widehat{=}$ ...
a_awake_send $\widehat{=}$ WHEN
  $a\_state = sleeping \wedge$
  $a\_out = IDL \wedge$
  $ab = IDL \wedge$
  $b\_out = IDL$
THEN
  $a\_state := sending \|$
  $a\_in := PN \|$
  $ab := PN \|$
  $case := \neg case$
END;
b_awake_send $\widehat{=}$ ...
a_awake_accept $\widehat{=}$ REFINES accept WHEN
  $a\_state = sleeping \wedge$
  $a\_out = PN \wedge$
  $b\_state = sending \wedge$
  $ab = IDL \wedge$
  $b\_out = IDL$
THEN
  $a\_state := accepting \|$
  $case := \neg case$
END;
b_awake_accept $\widehat{=}$ ...

- *a_send* représente l'émission initiale du signal PN. L'appareil n'a encore rien envoyé ni reçu. Notons que pour simplifier la preuve on place directement le signal à la deuxième étape de la transition : dans la variable *ab*.
- *ab_pass_out* fait avancer le signal le long du canal. *pass_out* permet la progression simultanée du signal dans les deux canaux.
- *a_accept* est déclenché à la réception d'un signal PN si celui-ci n'a été émis que d'un coté.

- a_sleep se produit lorsque l'appareil émet le signal PN et commence à le recevoir aussi. Dans ce cas, l'émission du signal retourne au signal de base IDL. C'est la découverte de la contention.
- a_awake_send est la ré-émission du signal après la contention. Il ne doit se déclencher qu'après la remise à zéro (signal IDL) de tout le canal d'émission.
- a_awake_accept est l'acceptation d'un signal reçu après la phase de contention. Cet évènement doit aussi vérifier que le canal d'émission a bien été remis à zéro. Nous trouvons aussi dans la garde que l'autre appareil doit bien être dans l'état *sending*, si ce n'est pas le cas il aurait été possible d'accepter un signal PN en cours d'effacement.

Les conditions décrites ci-dessus dans la garde des deux derniers évènements sont particulières car elles utilisent des informations qui ne sont pas locales à l'appareil seul. Ces conditions sont en fait la spécification de ce qui devra être raffiné par la propriété temps-réel que nous allons introduire.

3.5 Troisième modèle : temps de propagation

Dans ce raffinement, nous ajoutons un temps de propagation précis pour la propagation d'un signal au sein d'un canal. Ce temps sera la constante *prop*, elle prend ses valeurs dans $\mathbb{N}$ privé de 0. Nous allons appliquer notre patron pour ajouter la contrainte de temps, qui est le délai *prop* entre le début de l'émission d'un nouveau signal et le début de sa réception. Soit a et b deux appareils distincts, comme ensemble ACT d'actions nous avons a_pass et b_pass. Il s'agit bien de l'action *pass* mais comme nous somme dans un système distribué, il y a une version de cette action pour chaque appareil.

Au lieu d'utiliser directement la fonction at du patron, on peut faire un raffinement de donnée avec $at(a\_pass) = at\_a\_pass$ et $at(b\_pass) = at\_b\_pass$. On obtient alors deux-sous ensembles de $\mathbb{N}$ au lieu d'une fonction.

3.5.1 Invariant

Le typage des nouvelles variables est $(time \in \mathbb{N})$ et $(at\_a\_pass \subseteq \mathbb{N})$ et $(at\_b\_pass \subseteq \mathbb{N})$. L'invariant du patron doit être appliqué aux deux ensembles d'agenda :

$at\_a\_pass \cup at\_b\_pass \neq \varnothing \Rightarrow time \leq \min(at\_a\_pass \cup at\_b\_pass)$

Comme at_a_pass représente la date de réception d'un nouveau signal, et que les changements de signaux mettent du temps pour se propager (le temps de propagation), alors les valeurs de cet ensemble sont bornées par $time + prop$:

$\forall x \cdot (x \in at\_a\_pass \Rightarrow x \leq time + prop)$

L'ensemble at_a_pass est fini et sa cardinalité reflète le nombre de changement de signaux sur les canaux. Dans le modèle, nous utilisons une formulation équivalente avec des quantifications plutôt que des cardinalités car cela est plus efficace avec les logiciels impliqués dans la preuve interactive.

$b\_in = ba \wedge ba = a\_out \Leftrightarrow at\_a\_pass = \varnothing$

$a\_in = ab \wedge ab \neq b\_out \Leftrightarrow card(at\_a\_pass) = 1$

$b\_in \neq ba \wedge ba \neq a\_out \Leftrightarrow card(at\_a\_pass) = 2$

Un appareil ne peut pas commencer à émettre après la réception d'un signal PN, nous avons donc :

$\forall (x, y) \cdot (x \in at\_a\_pass \wedge y \in at\_b\_pass \Rightarrow |x - y| < prop)$

Si la cardinalité de at_b_pass est de deux, alors la différence de ces éléments est strictement inférieure à *prop* car ils sont bornés par $time+prop$ et que l'action *pass* se déclenche prioritairement aux évènements de type *sleep*.

$\forall (x, y) \cdot (x \in at\_b\_pass \wedge y \in at\_b\_pass \Rightarrow |x - y| < prop)$

Si un appareil reçoit et émet le signal PN alors celui-ci ne vient pas de commencer à émettre ce signal à cet instant :

$b\_in = PN \wedge b\_out = PN \Rightarrow time + prop \notin at\_a\_pass$

L'évènement *pass* se déclenche prioritairement à *send* :

$time \in at\_a\_pass \cup at\_b\_pass \Rightarrow time + prop \notin at\_a\_pass \cup at\_b\_pass$

Et enfin, cette partie de l'invariant est directement utilisée dans la preuve de raffinement de *ab_pass_out* :

$$ab{\neq}b\_out \wedge time \in at\_b\_pass - at\_a\_pass \Rightarrow b\_state{\neq}sending \vee b\_out = IDL$$

3.5.2 Évènements

Pour exprimer une priorité entre deux évènements (quand ceux-ci sont activables dans le même état), on peut ajouter une garde qui bloque l'évènement que l'on veut retarder. Ceci est beaucoup utilisé dans ces évènements pour donner la priorité à l'environnement des appareils (comme la gestion des canaux) plutôt qu'aux appareils.

Par soucis de lisibilité, nous allons définir ce modèle uniquement par les différences qu'il possède avec le modèle qu'il raffine. Cela permet de voir immédiatement ce qu'implique le raffinement. Pour cela, les nouvelles lignes seront marqué d'un $\oplus$ et les lignes supprimées d'un $\ominus$. Toutes les nouvelles lignes dans les gardes sont connectées par $\wedge$ et toutes les lignes des actions sont connectées par $\|$.

init $\widehat{=}$ BEGIN
  $\oplus time := 0$
  $\oplus at\_a\_pass, at\_b\_pass := \varnothing, \varnothing$ END;
a_send $\widehat{=}$ WHEN
  $\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
 THEN
  $\oplus at\_b\_pass := at\_b\_pass \cup \{time + prop\}$
 END;
b_send $\widehat{=}$ ...
ab_pass_out $\widehat{=}$ WHEN
  $\ominus(b\_state \neq sending \vee b\_out \neq PN)$
  $\oplus time \in at\_b\_pass - at\_a\_pass$
 THEN
  $\oplus at\_b\_pass := at\_b\_pass - \{time\}$ END;
ba_pass_out $\widehat{=}$ ...
pass_out $\widehat{=}$ WHEN
  $\oplus time \in at\_a\_pass \cap at\_b\_pass$
 THEN
  $\oplus at\_a\_pass := at\_a\_pass - \{time\}$
  $\oplus at\_b\_pass := at\_b\_pass - \{time\}$ END;
a_accept $\widehat{=}$ WHEN
  $\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
 THEN ... END;
b_accept $\widehat{=}$ ...
a_sleep $\widehat{=}$ WHEN
  $\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
 THEN
  $\oplus at\_b\_pass := at\_b\_pass \cup \{time + prop\}$
 END;
b_sleep $\widehat{=}$ ...
a_awake_send $\widehat{=}$ WHEN
  $\oplus time \notin (at\_a\_pass \cup at\_b\_pass)$
 THEN
  $\oplus at\_b\_pass := at\_b\_pass \cup \{time + prop\}$
 END;
b_awake_send $\widehat{=}$ ...
a_awake_accept $\widehat{=}$ WHEN
  $\oplus time \notin (at\_a\_pass \cup at\_b\_pass)\wedge$
 THEN ... END;
b_awake_accept $\widehat{=}$ ...
tick_tock $\widehat{=}$
 ANY tm WHERE
  $tm \in \mathbb{N} \wedge tm > time\wedge$
  $((at\_a\_pass \cup at\_b\_pass) \neq \varnothing \Rightarrow$
  $tm \leq \min(at\_a\_pass \cup at\_b\_pass))\wedge$
  $(a\_state \neq sending \vee a\_out \neq PN)\wedge$
  $(b\_state \neq sending \vee b\_out \neq PN)$
 THEN $time := tm$ END;

3.6 Quatrième modèle : temps d'attente

Ce dernier raffinement enlève toutes les conditions abstraites présentes dans les gardes et les remplace par l'utilisation des deux délais temps-réel. Nous avons donc deux nouvelles constantes : *st* (*short time*) et *lt* (*long time*) qui sont des nombres entiers non nuls. Pour que le protocole se déroule correctement, leurs valeurs doivent respecter :

$st \geq prop \times 2$

et

$lt \geq prop \times 2 + st - 1$

On étend l'utilisation du patron par deux nouvelles valeurs pour l'ensemble ACT avec a_awake, b_awake. Comme précédemment nous définissons : $at\_a\_awake = at(a\_awake)$ et $at\_b\_awake = at(b\_awake)$. Enfin, les deux variables supplémentaires a_sleept et b_sleept permettent de noter le délai choisi par chaque appareil.

On peut voir dans la figure 3.4 un diagramme représentant une situation typique de contention (avec les valeurs $prop = 3$, $st = 6$ and $lt = 11$). L'élection sera un succès si les délais choisis sont différents. On peut voir dans la figure que, dans ce cas, l'appareil qui arrête son attente en premier, possède suffisamment de temps pour que son signal parvienne à l'autre appareil.

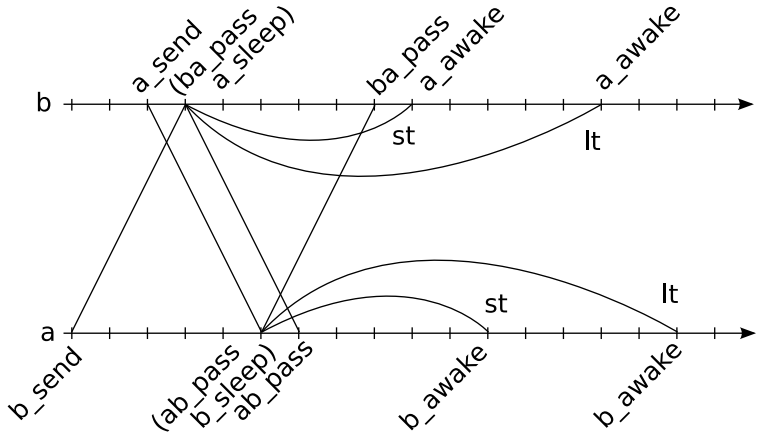

FIG. 3.4 – Timeline

3.6.1 Invariant

De la même manière que précédemment, nous abordons ici uniquement le cas de a, celui de b se déduisant par symétrie.

Les deux ensembles at_a_awake et at_b_awake sont des sous-ensembles d'entiers. Les valeurs des variables a_sleept et b_sleept sont dans $\{st, lt\}$. Nous avons bien sur le même invariant exprimant que la valeur de $time$ doit être inférieure ou égale à celle contenue dans la fonction at.

Il y a une et une seule valeur dans at_a_awake si et seulement si l'état de a est $sleeping$:

$a\_state \neq sleeping \Leftrightarrow at\_a\_awake = \varnothing$

$a\_state = sleeping \Leftrightarrow card(at\_a\_awake) = 1$

Et cette valeur est bornée :

$\forall x \cdot (x \in at\_a\_awake \Rightarrow x \leq time + a\_sleept)$

$(time \in at\_a\_awake \wedge a\_sleept = b\_sleept \Rightarrow$
$\quad \forall x \cdot (x \in at\_b\_awake \Rightarrow x < time + prop))$

Nous pouvons aussi identifier plusieurs bornes inférieures, suivant les cas :

$(a\_in = PN \wedge (a\_out = PN \vee (a\_out = IDL \wedge ba = PN)) \Rightarrow$
$\quad \forall x \cdot (x \in at\_b\_awake \Rightarrow time + b\_sleept - prop < x))$

$a\_in = PN \wedge a\_out = PN \Rightarrow \forall x \cdot (x \in at\_b\_awake \Rightarrow time + prop < x)$

$case = \text{FALSE} \wedge b\_state = sending \Rightarrow$
$\quad \forall x \cdot (x \in at\_a\_awake \Rightarrow time + prop < x)$

$(time \in at\_a\_awake \wedge a\_sleept \neq b\_sleept \Rightarrow$
$\quad \forall x \cdot (x \in at\_b\_awake \Rightarrow time + prop \leq x))$

Nous savons que la valeur $time + prop$ n'est pas dans at_a_awake sous une certaine condition :

$ab = PN \wedge b\_out = IDL \Rightarrow time + prop \notin at\_a\_awake$

Ici nous pouvons voir que, dans deux différents cas, le changement de signal est reçu avant l'écoulement du délai :

$case = \text{FALSE} \Rightarrow \forall (x, y) \cdot (x \in at\_a\_pass \wedge y \in at\_a\_awake \Rightarrow x < y)$

$a\_sleept \neq b\_sleept \Rightarrow \forall (x, y) \cdot (x \in at\_a\_pass \wedge y \in at\_a\_awake \Rightarrow x \leq y)$

Et dans la condition suivante, l'écoulement du délai a lieu cette fois avant la réception du changement de signal :

$(case = \text{TRUE} \wedge a\_sleept = b\_sleept \Rightarrow$
$\quad \forall (x, y) \cdot (x \in at\_a\_awake \wedge y \in at\_a\_pass \Rightarrow x < y))$

Après la découverte de la contention, l'appareil a a remis à zéro son canal un temps de propagation avant l'écoulement du délai :

$\forall (x, y) \cdot (x \in at\_a\_pass \wedge y \in at\_b\_awake \Rightarrow x + prop \leq y)$

Si les délais choisis sont égaux, alors les appareils n'ont pas le temps de transmettre un signal complètement afin de réaliser l'élection :

$(a\_sleept = b\_sleept \Rightarrow$

$\forall(x,y)\cdot(x \in at\_a\_awake \wedge y \in at\_b\_awake \Rightarrow |x-y| < prop))$

Si les délais sont différents, alors un changement de signal est complètement transmis. Ce qui permet à l'élection de se réaliser :

$(a\_sleept \neq b\_sleept \Rightarrow$
$\forall(x,y)\cdot(x \in at\_a\_awake \wedge y \in at\_b\_awake \Rightarrow prop \leq |x-y|))$

Si la cardinalité de at_a_pass est de deux, alors il y a un temps de $b\_sleept - prop$ entre la fin du délai et la réception du premier changement de signal :

$(b\_in \neq ba \wedge ba \neq a\_out \Rightarrow$
$\forall x\cdot(x \in at\_b\_awake \Rightarrow \min(at\_a\_pass) + b\_sleept - prop < x))$

Finalement, ces dernières formules nous donne les éléments permettant de prouver le raffinement de a_awake_send et a_awake_accept :

$time \in at\_a\_awake \Rightarrow ab = IDL \wedge b\_out = IDL$
$time \in at\_a\_awake \wedge (a\_out = PN \vee ba = PN) \Rightarrow b\_state = sending$
$case = FALSE \wedge time \in at\_a\_awake \Rightarrow b\_state = sleeping$

3.6.2 Évènements

Comme précédemment, nous donnons seulement les différences introduites, les nouvelles sont marquées par $\oplus$ les lignes supprimées par $\ominus$. Si un évènement n'est pas présent, alors c'est qu'il n'y a pas de différence ou alors qu'il est symétrique avec celui présenté ici.

Avec les propriétés temps-réel des évènements de type *awake*, nous pouvons supprimer toutes les conditions abstraites des gardes des évènements. Ceci introduit des obligations de preuves qu'il faut prouver à l'aide de l'invariant pour montrer le raffinement.

init $\widehat{=}$ BEGIN
$\oplus at\_a\_awake, at\_b\_awake := \varnothing, \varnothing$
$\oplus a\_sleept, b\_sleept := st, st$ END;
a_sleep $\widehat{=}$
ANY $\oplus sleep$
WHERE $\oplus sleep \in \{st, lt\}$
THEN $\oplus at\_a\_awake :=$
$at\_a\_awake \cup \{time + sleep\}$
$\oplus a\_sleept := sleep$ END;
a_awake_send $\widehat{=}$ WHEN
$\oplus time \in at\_a\_awake$
$\ominus a\_state = sleeping$
$\ominus ab = IDL$
$\ominus b\_outs = IDL$
THEN
$\oplus at\_a\_awake := at\_a\_awake - \{time\}$ END;
a_awake_accept $\widehat{=}$ WHEN
$\oplus time \in at\_a\_awake$
$\ominus a\_state = sleeping$
$\ominus b\_state = sending$
$\ominus ab = IDL$
$\ominus b\_out = IDL$
THEN
$\oplus at\_a\_awake := at\_a\_awake - \{time\}$ END;
tick_tock $\widehat{=}$ ...
"même patron avec en plus at_a_awake et at_b_awake"

3.7 Conclusions

De nombreux travaux existent sur *IEEE 1394 Root Contention Protocol* (RCP). On peut trouver une synthèse de ces travaux dans [10]. Notre travail étend ces résultats avec une autre approche, tout en restant dans le cadre de la méthode B. Notre méthode de vérification est la preuve mécanisée par l'intermédiaire d'invariants et de raffinement entre plusieurs modèles. Il est clair qu'une preuve interactive demande bien plus de travail pour être menée à bien qu'une vérification par model-checking ou à l'aide de procédure de décision (des études de ce type sont référencées dans l'article de synthèse). Mais cela permet d'utiliser un langage général et expressif. Ainsi, on peut mener la preuve dans le cadre le plus général c'est-à-dire de laisser indéterminée les constantes de délai temps-réel *st* et *lt* avec des hypothèses de bonne définition. Une fois les invariants trouvés, la preuve est répétitive et nous pensons que beaucoup d'améliorations peuvent être faites sur le prouveur (ici B4Free).

Nous avons montré qu'avec notre patron, il est possible de modéliser un système temps-réel au sein de la méthode B. La relation de raffinement permet d'introduire les contraintes temps-réel de manière incrémentale. Et à chaque raffinement, nous pouvons vérifier et valider le modèle obtenu.

La preuve obtenue concerne avant tout la correction partielle mais comme l'invariant exprime des propriétés temps-réel, nous avons des éléments sur le comportement dynamique du système.

En particulier, on peut voir que si les délais choisis sont différents ($a\_sleept \neq b\_sleept$), alors le signal d'élection a le temps de parvenir à son destinataire :

$$\forall(x,y)\cdot(x \in at\_a\_awake \wedge y \in at\_b\_awake \Rightarrow prop \leq |x-y|))$$

Mais si les délais sont égaux ($a\_sleept = b\_sleept$), les appareils n'auront pas le temps de communiquer :

$$\forall(x,y)\cdot(x \in at\_a\_awake \wedge y \in at\_b\_awake \Rightarrow |x-y| < prop))$$

où at_a_awake et at_b_awake sont des sous-ensembles d'entiers représentant à quel moment les appareils vont arrêter d'attendre, et $prop$ est le temps de propagation nécessaire pour qu'un changement de signal traverse le canal entre deux appareils.

Chapitre 4

Étude de cas : algorithme de Simpson

4.1 Introduction

Nous allons développer par la preuve un algorithme de communication asynchrone, il s'agit d'un algorithme de Simpson [9] dans une version à deux emplacements mémoire. Cette version n'est pas totalement asynchrone, mais elle nécessite moins d'espace mémoire que la version à 4 emplacements mémoire. Comme l'asynchronisme n'est pas total, certains ordonnancements des processus communicants seront interdits, pour spécifier cela nous allons utiliser des contraintes temps-réel. Et plus précisement nous allons prendre en compte les durées de certaines propriétés du système formé par l'algorithme.

Nous allons commencer dans la section 4.2 par spécifier l'algorithme et les propriétés qu'il présente. Nous commencerons à spécialiser ces propriétés dans la section 4.3 et les premiers éléments, comme la mémoire, apparaissent dans la section 4.4. La section 4.5 continue à introduire l'implémentation et étudie une propriété importante sur l'ordonnancement du système. Enfin, la section 4.6 utilise le patron des compteurs de durée pour développer les propriétés temps-réel de la description finale de l'algorithme.

4.2 Premier modèle : spécification de l'algorithme

Le but de l'algorithme est de permettre une communication unidirectionnelle asynchrone entre deux entités. La communication se fait dans un seul sens, nous nommons une des entités l'écrivain et l'autre le lecteur. De plus, le sens de communication va de l'écrivain vers le lecteur. À tout moment l'écrivain peut envoyer une nouvelle valeur, et le lecteur peut en prendre connaissance, ou non, de manière complètement indépendante. Ceci se fait grâce à des variables, une mémoire, partagées entre les deux acteurs.

A titre d'exemple on peut imaginer que l'écrivain est un thermomètre électronique mettant régulièrement la température mesurée à jour et que le lecteur est un autre appareil lisant la valeur courante de la température quand il en a besoin.

Dans l'implémentation, les opérations de lecture et d'écriture ne sont pas atomiques. Pour le moment, nous allons commencer le développement de ce système à un niveau plus abstrait en considérant un évènement atomique de lecture nommé *read*, et un d'écriture nommé *write*.

Pour représenter cette particularité à ce niveau d'abstraction, il va exister un décalage entre la valeur lue et la valeur écrite. Mais l'algorithme donne des garanties sur le niveau de fraîcheur de la valeur lue et nous allons formaliser cela dans la suite.Informellement, nous pouvons dire que la valeur lue est au moins aussi récente que la dernière valeur écrite au moment de la lecture précédente.

La figure 4.1 illustre le comportement de cet algorithme. Il s'agit d'une séquence de cinq écritures symbolisées par $w1, w2, ..., w5$ et de sept lectures symbolisées par $r1, r2, ..., r7$. Soit $DATA$ l'ensemble des données que l'algorithme va manipuler. Les éléments de $DATA$ seront notés : $d1, d2, d3, ...$ La valeur écrite ou lue est donnée entre parenthèses. On peut observer en lecture $r3$ ce décalage : la valeur lue est $d1$ et non pas $d2$. Par contre la lecture $r4$ doit être obligatoirement à jour et lire

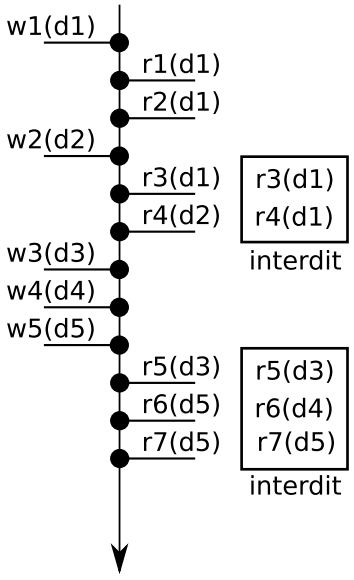

FIG. 4.1 – Séquence de lecture et d'écriture

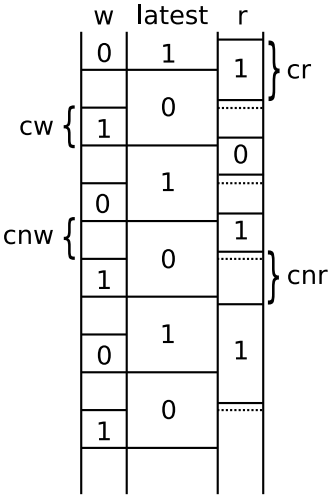

FIG. 4.2 – Évolution de l'algorithme

la valeur $d2$. Le même scénario est visible pour les lectures $r5$ à $r7$, avec un exemple de scénario interdit dans l'encadré.

SETS

DATA

CONSTANTS

$d1$

AXIOMS

$axm1 : d1 \in DATA$

END

VARIABLES

wn Write Number

rn Read Number

r_at Read AT

lw_at Last Write AT

rv Read Values

wv Written Values

Variables et typage Dans ce modèle $m0$ nous allons numéroter les opérations d'écritures avec : la variable wn (Write Number) qui est le nombre d'écritures effectuées ($typ1 : wn \in \mathbb{N}_1$) ; et la variable rn (Read Number) qui est le nombre de lectures effectuées ($typ2 : rn \in \mathbb{N}_1$). La variable rv (Read Values) permettra de connaître la valeur $rv(x)$ lu à la lecture numéro x ($typ3 : rv \in 1..rn \rightarrow DATA$). De même, la variable wv (Written Values) permettre de connaître la valeur $wv(x)$ écrite à l'écriture numéro x ($typ4 : wv \in 1..wn \rightarrow DATA$). Les variables rv et wv sont des «traces» qui conservent tout l'historique de la valeur lue et écrite.

Les deux dernières variables permettrons, entre autre, de quantifier le décalage entre les valeurs lues et écrites. La variable r_at (Read AT) est similaire à la variable rv mais donne l'indice $r\_at(x)$ (correspondant au numéro de l'écriture) de la valeur $rv(x)$ pour la lecture x ($typ5 : r\_at \in 1..rn \rightarrow 1..wn$). Et enfin la variable lw_at (Last Write AT) donne $lw\_at(x)$ l'indice de la dernière écriture au moment de la lecture x ($typ6 : lw\_at \in 1..rn \rightarrow 1..wn$).

INVARIANTS

$typ1 : wn \in \mathbb{N}_1$

$typ2 : rn \in \mathbb{N}_1$

$typ3 : rv \in 1 \mathinner{..} rn \rightarrow DATA$

$typ4 : wv \in 1 \mathinner{..} wn \rightarrow DATA$

$typ5 : r\_at \in 1 \mathinner{..} rn \rightarrow 1 \mathinner{..} wn$

$typ6 : lw\_at \in 1 \mathinner{..} rn \rightarrow 1 \mathinner{..} wn$

$inv1 : \forall i \cdot i \in 1 \mathinner{..} rn \Rightarrow r\_at(i) \leq lw\_at(i)$

$inv2 : \forall i \cdot i \in 1 \mathinner{..} rn - 1 \Rightarrow lw\_at(i) \leq r\_at(i+1)$

$inv3 : wn \geq lw\_at(rn)$

$inv4 : rv = r\_at; wv$

$inv5 : wv(1) = d1$

Initialisation

begin

$act1 : wn := 1$

$act2 : rn := 1$

$act3 : r\_at := \{1 \mapsto 1\}$

$act4 : lw\_at := \{1 \mapsto 1\}$

$act5 : rv := \{1 \mapsto d1\}$

$act6 : wv := \{1 \mapsto d1\}$

end

Invariant Nous pouvons maintenant étudier les propriétés formulées dans les invariants du modèle (en fait, les formules de typage $typ1...typ6$ sont aussi des invariants). La première assertion affirme qu'il est possible d'avoir un décalage $lw\_at(i) - r\_at(i)$ non nul ($inv1 : \forall i \cdot\ i \in 1 \mathinner{..} rn \Rightarrow r\_at(i) \leq lw\_at(i)$). Par exemple sur la figure, la lecture $r3$ à pour numéro $i = 3$ avec $lw\_at(3) = 2$ pour la dernière écriture, mais $r\_at(3) = 1$ pour l'indice réellement lu. Mais nous savons que la valeur lue est au moins aussi fraîche que la dernière valeur écrite au moment de la lecture d'avant ($inv2 : \forall i \cdot\ i \in 1 \mathinner{..} rn - 1 \Rightarrow lw\_at(i) \leq r\_at(i+1)$). On peut aussi dire que le numéro wn de la dernière écriture peut être supérieure, et ici sans limite, à la dernière valeur écrite à la dernière lecture rn ($inv3 : wn \geq lw\_at(rn)$), il ne faut pas oublier que l'algorithme admet un nombre quelconque d'écriture entre deux lectures (et vice-versa). Les valeurs lues sont bien celles qui ont été écrite mais décalées suivant les indices contenus dans r_at ($inv4 : rv = r\_at; wv$). Enfin le dernier invariant n'est utile que pour se rappeler une valeur donnée à l'initialisation ($inv5 : wv(1) = d1$).

Initialisation L'initialisation met en place une écriture et une lecture initiale. Les actions de l'initialisation sont donc : Mettre le nombre d'écriture et de lecture $act1 : wn := 1$ et $act2 : rn := 1$; Initialiser deux séquences d'indices avec l'indice initial $act3 : r\_at := \{1 \mapsto 1\}$ et $act4 : lw\_at := \{1 \mapsto 1\}$; Et poser, arbitrairement à $d1$, la première valeur manipulée avec $act5 : rv := \{1 \mapsto d1\}$ et $act6 : wv := \{1 \mapsto d1\}$.

Event read $\widehat{=}$

any

ri Read Index

where

$grd1 : ri \in lw\_at(rn) \mathinner{..} wn$

then

$act1 : rn := rn + 1$

$act2 : r\_at(rn+1) := ri$

$act3 : lw\_at(rn+1) := wn$

$act4 : rv(rn+1) := wv(ri)$

end

Event write $\widehat{=}$

any

d

where

$grd1 : d \in DATA$

then

$act1 : wn := wn + 1$

$act2 : wv(wn+1) := d$

end

Évènement *read* L'évènement *read* représente l'action du lecteur, *read* possède une variable locale ri (Read Index) qui représente l'indice de la valeur lue parmi les valeurs écrites wv.

Garde Cette variable locale et l'évènement doivent respecter la garde $grd1 : ri \in lw\_at(rn)\,..\,wn$ qui précise le décalage possible entre les indices des valeurs lues et écrites, il faut, évidement, que l'indice lu soit inférieur à wn, mais aussi qu'il soit au moins égal à l'indice écrit lors de la dernière lecture. Ceci assure que le lecteur va bien progresser entre chaque lecture. Mais il se peut que la valeur lue ne soit pas la dernière car il peut y avoir eu des nouvelles écritures depuis la dernière lecture.

Action L'action réalisée par l'évènement *read* est sans surprise, il s'agit : d'incrémenter le nombre de lecture effectuée $act1 : rn := rn+1$; de noter quel indice est lu $act2 : r\_at(rn+1) := ri$; de noter l'indice de la dernière lecture $act3 : lw\_at(rn+1) := wn$; et enfin d'ajouter le résultat de la lecture dans la séquence des valeurs lues $act4 : rv(rn+1) := wv(ri)$.

Évènement *write* L'évènement *write* représente l'action de l'écrivain. Ce qui consiste à prendre une valeur d (sous la forme d'une variable local) dans l'ensemble de donnée $grd1 : d \in DATA$ et ensuite à la rajouter dans la séquence de valeurs écrites $act2 : wv(wn+1) := d$ tout en incrémentant le nombre d'écriture effectuées $act1 : wn := wn + 1$.

4.3 Deuxième modèle

Ce modèle $m1$ sert à raffiner les notations de la spécification générale du modèle $m0$.

VARIABLES

wn

wv

rr Read Result

lw_at_lr Last Write AT Last Read

INVARIANTS

$typ1 : rr \in DATA$

$typ2 : lw\_at\_lr \in 1\,..\,wn$

$inv1 : rv(rn) = rr$

$inv2 : lw\_at\_lr = lw\_at(rn)$

Variables et typage Parmi les variables de $m0$ nous allons garder wn etwv, les variables rn, r_at, lw_at et rv disparaissent au profit des nouvelles variables rr (Read Result) et lw_at_lr (Last Write AT Last Read). La variable rr représente le résultat de la dernière lecture $typ1 : rr \in DATA$. La variable lw_at_lr représente l'indice de la dernière écrite au moment de la dernière lecture $typ2 : lw\_at\_lr \in 1 \mathinner{..} wn$.

Invariant Pour raffiner le modèle il faut, en particulier, formaliser le lien entre les nouvelles variables et celles qui ont disparues. Ce qui permet aussi de formaliser la signification des ces nouvelles variables, nous retrouvons donc l'explication des paragraphes précédents avec $inv1 : rv(rn) = rr$ et $inv2 : lw\_at\_lr = lw\_at(rn)$.

Initialisation

begin

$act1 : wn := 1$

$act2 : wv := \{1 \mapsto d1\}$

$act3 : rr := d1$

$act4 : lw\_at\_lr := 1$

end

Initialisation L'initialisation est similaire à celle du modèle abstrait avec : le nombre d'écritures à $act1 : wn := 1$; une valeur $d1$ dans la séquence des valeurs écrites $act2 : wv := \{1 \mapsto d1\}$; on initialise le résultat de la lecture $act3 : rr := d1$; et le dernier indice écrit est 1 $act4 : lw\_at\_lr := 1$.

Event read $\widehat{=}$

Refines read

any

ri

where

$grd1 : ri \in lw\_at\_lr \mathinner{..} wn$

then

$act1 : lw\_at\_lr := wn$

$act2 : rr := wv(ri)$

end

Event write $\widehat{=}$

Refines write

any

d

where

$grd1 : d \in DATA$

then

$act1 : wn := wn + 1$

$act2 : wv(wn + 1) := d$

end

Évènement *read* Cet évènement raffine l'évènement abstrait du même nom. Il garde la même variable ri. La garde devient $grd1 : ri \in lw\_at\_lr \mathinner{..} wn$. Il ne reste plus que deux actions : celle qui permet de noter le dernier indice écrit à ce moment là devient $act1 : lw\_at\_lr := wn$ et le résultat de la lecture devient $act2 : rr := wv(ri)$.

Évènement *write* Cet évènement est identique à l'évènement abstrait du même nom.

4.4 Troisième modèle : éléments de l'implémentation

Ce modèle $m2$ qui raffine le modèle $m1$ replace les opérations de lecture ou d'écriture atomique par des opérations non-atomiques. Pour cela nous allons considérer deux évènements : un pour le début de l'opération et un pour la fin. De plus ce raffinement parmet d'ajouter la gestion de la mémoire utilisé par l'algorithme. Cette mémoire a deux emplacements puisque nous allons étudier la version de l'algorithme de Simpson avec deux slots.

VARIABLES

wn

wv

rr

$latest$

$reading$

$writing$

mem

INVARIANTS

$typ1 :\ mem \in 0\,..\,1 \rightarrow \mathbb{N}$

$typ2 :\ latest \in 0\,..\,1$

$typ3 :\ reading \in \mathbb{P}(0\,..\,1)$

$typ4 :\ writing \in \mathbb{P}(0\,..\,1)$

$inv1 :\ reading \cap writing = \varnothing$

$inv2 :\ latest \notin writing$

$inv3 :\ mem(latest) = wn$

$inv4 :\ mem(1 - latest) = wn - 1$

$inv5 :\ \forall x, y \cdot x \in reading \wedge y \in reading \Rightarrow x = y$

$inv6 :\ \forall x, y \cdot x \in writing \wedge y \in writing \Rightarrow x = y$

$inv7 :\ reading \neq \varnothing \wedge latest \notin reading \Rightarrow lw\_at\_lr \leq wn - 1$

$inv8 :\ wn = 1 \Rightarrow latest = 1 \wedge reading \subseteq \{1\}$

Variables et typage La variable lw_at_lr disparait et nous conservons les variables wn, wv et rr. Nous ajoutons ensuite les variables : mem qui représente les indices des deux valeurs enregistrées dans la mémoire $typ1 : mem \in 0..1 \rightarrow \mathbb{N}$; $latest$ qui est le pointeur sur l'emplacement de la mémoire contenant l'indice de la dernière valeur écrite $typ2 : latest \in 0\,..\,1$; $reading$ qui est l'ensemble des emplacements mémoire en cours de lecture $typ3 : reading \in \mathbb{P}(0\,..\,1)$; $writing$ fait de même pour les écritures en cours $typ4 : writing \in \mathbb{P}(0\,..\,1)$.

On peut voir sur la figure 4.2 page 30 un scénario possible pour l'évolution de l'algorithme. Le sens de l'exécution est du haut vers le bas. La colonne w figure les valeurs à l'intérieur de l'ensemble $writing$, on voit que l'on alterne entre les opérations d'écriture et de repos et le système alterne aussi entre les emplacements écrits. La colonne $latest$ figure la variable du même nom et a chaque fin d'écriture elle est mise à jour. La colonne r représente elle $reading$. L'emplacement lu est égale à la valeur de $latest$ au début de la lecture. Les pointillés dans cette colonne représentent la limite que ne doit pas franchir une lecture sous peine de lire au même endroit qu'une écriture en cours. Les éléments en accolade ne sont pas encore utilisés à ce niveau du développement.

Invariants La propriété la plus importante que doit vérifier cette implémentation est qu'il n'est pas autorisé de lire et d'écrire en même temps sur le même emplacement mémoire, pour vérifier cela nous avons l'invariant : $inv1 : reading \cap writing = \varnothing$. Il faut aussi que l'écrivain n'écrase pas la dernière valeur qu'il a écrite : $inv2 : latest \notin writing$. En fait l'écrivain va alterner entre les deux emplacements mémoire, nous avons donc $inv3 : mem(latest) = wn$ et $inv4 : mem(1 - latest) = wn - 1$. Comme il n'y a qu'un écrivain et qu'un lecteur il ne peut y avoir au plus qu'une lecture ou

écriture en cours. Pour formaliser cela, nous évitons d'utiliser directement l'opérateur de cardinalité qui pose des problèmes avec le prouveur utilisé, nous avons donc : $inv5 : \forall x, y \cdot x \in reading \land y \in reading \Rightarrow x = y$ et $inv6 : \forall x, y \cdot x \in writing \land y \in writing \Rightarrow x = y$. Si on considère que le système effectue une lecture et qu'une écriture a été effectuée depuis le début de la lecture en cours alors l'indice de la dernière écriture au moment de la lecture précédente est au plus l'indice de la dernière écriture moins un : $inv7 : reading \neq \varnothing \land latest \notin reading \Rightarrow lw\_at\_lr \leq wn - 1$. Enfin le dernier invariant est utile pour rappeler de quelle manière le système a été utilisé $inv8 : wn = 1 \Rightarrow latest = 1 \land reading \subseteq \{1\}$.

Initialisation

begin

$act1 : wn := 1$

$act2 : wv := \{1 \mapsto d1\}$

$act3 : rr := d1$

$act4 : latest := 1$

$act5 : reading := \varnothing$

$act6 : writing := \varnothing$

$act7 : mem := \{0 \mapsto 0, 1 \mapsto 1\}$

end

Initialisation Nous avons les actions déjà présentes dans le modèle abstrait ($act1 : wn := 1$, $act2 : wv := \{1 \mapsto d1\}$ et $act3 : rr := d1$). Le système place, arbitrairement, le pointeur du dernier emplacement mémoire à avoir été écrit sur le premier $act4 : latest := 1$. Il n'y a pas, initialement, d'opérations en cours : $act5 : reading := \varnothing$ et $act6 : writing := \varnothing$. Enfin la mémoire est initialisée à wn et $wn - 1$: $act7 : mem := \{0 \mapsto 0, 1 \mapsto 1\}$.

Event begin_read $\widehat{=}$

when

$grd1 : reading = \varnothing$

then

$act1 : reading := \{latest\}$

end

Event end_read $\widehat{=}$

Refines read

any

ri

x

where

$grd1 : x \in reading$

$grd2 : ri = mem(x)$

then

$act1 : reading := \varnothing$

$act2 : rr := wv(ri)$

end

Évènement *begin_read* Ce nouvel évènement représente le début de l'opération de lecture

Garde Il est possible de commencer l'opération de lecture si le lecteur n'est pas déjà en train de lire : $grd1 : reading = \varnothing$.

Action Pour lire dans la mémoire, il s'agit de regarder quel est le dernier emplacement qui a été écrit et de le lire : $act1 : reading := \{latest\}$.

Évènement *end_read* Cet évènement raffine *read*, il est la fin de la lecture et hérite donc, par le raffinement de *read*, de la réalisation de l'opération de lecture. L'évènement conserve la variable ri de l'abstrait et ajoute la variable x qui représente le pointeur de l'emplacement mémoire en cours de lecture.

Garde Les gardes sont donc $grd1 : x \in reading$, qui requiert implicitement que l'ensemble $reading$ soit non vide, et $grd2 : ri = mem(x)$, qui positionne l'indice ri à celui stocké dans la mémoire à l'emplacement en cours de lecture.

Action L'évènement termine la lecture en cours par l'action $act1 : reading := \varnothing$ et réalise la lecture en réenregistrant la vième valeur de la séquence des écritures wv dans rr avec l'action $act2 : rr := wv(ri)$.

Event begin_write $\widehat{=}$
when
$grd1 : writing = \varnothing$
$grd2 : reading \neq \varnothing \Rightarrow latest \in reading$
then
$act1 : writing := \{1 - latest\}$
end

Event end_write $\widehat{=}$
Refines write
any
d
x
where
$grd1 : d \in DATA$
$grd2 : x \in writing$
then
$act1 : wn := wn + 1$
$act2 : wv(wn + 1) := d$
$act3 : writing := \varnothing$
$act4 : mem(x) := mem(latest) + 1$
$act5 : latest := x$
end

Évènement *begin_write* Ce nouvel évènement représente le début de la lecture.

Garde Il ne doit pas y avoir déjà d'opération de lecture en cours $grd1 : writing = \varnothing$. Nous devons maintenant introduire la partie la plus spécifique de cet algorithme. En effet l'algorithme de Simpson à deux emplacements mémoire est une version dégénérée car elle ne permet pas une communication complètement asynchrone. Mais en contre partie l'empreinte mémoire est réduite. Pour que cela fonctionne il faut, si une lecture est déjà en cours, ne pas avoir déjà réalisé une écriture depuis. Nous exprimons cela par la garde $grd2 : reading \neq \varnothing \Rightarrow latest \in reading$.

Actions L'action du début de l'écriture consiste à déclarer l'opération d'écriture par $act1 : writing := \{1 - latest\}$. L'emplacement mémoire choisit $1 - latest$ est, parmi les deux possibles, celui qui n'est pas *latest*. L'écriture va donc alterner entre les deux emplacements.

Évènement *end_write* Cet évènement raffine *write* du modèle précédent et représente la fin de l'opération d'écriture. La variable locale d est conservée du modèle abstrait et la variable x est ajoutée, elle représente l'emplacement mémoire en cours d'écriture.

Garde La garde $grd1 : d \in DATA$ est la même depuis le début. Quand à la garde $grd2 : x \in writing$, elle permet de fixer x à l'emplacement en cours de lecture.

Action Les deux actions $act1 : wn := wn + 1$ et $act2 : wv(wn + 1) := d$ étaient déjà présentes dans les modèles plus abstraits. L'action $act3 : writing := \varnothing$ déclare l'arrêt de l'écriture et l'action $act4 : mem(x) := mem(latest) + 1$ change effectivement l'emplacement mémoire avec le nouvel indice qui est en fait $wn + 1$. Enfin le système met à jour la variable qui indique quel emplacement mémoire est le plus à jour avec l'action $act5 : latest := x$.

4.5 Quatrième modèle : introduction du buffer et ordonnancement

Dans ce raffinement du modèle $m2$, nous allons introduire un buffer contenant les valeurs écrites plutôt que les indices. La clé pour cet algorithme est de ne pas écrire deux fois de suite pendant une même lecture, cela provoquerait une écriture à un emplacement en cours de lecture, ce qui est indésirable. Pour spécifier cela nous allons compter combien de fois l'écrivain s'active pendant une lecture et donner les conditions nécessaires pour que l'algorithme fonctionne suivant ce compteur. Le modèle possède donc deux nouvelles variables : buf, de type $typ1 : buf \in 0 \mathinner{..} 1 \rightarrow DATA$, pour le buffer et *count*, de type $typ2 : count \in \mathbb{N}$, pour le compteur. La variable buf va permettre de remplacer wn, wv et mem qui disparaissent donc au profit de cette variable plus proche de l'implémentation.

VARIABLES

rr
$latest$
$reading$
$writing$
buf Buffer
$count$

INVARIANTS

$typ1 : buf \in 0 \mathinner{..} 1 \rightarrow DATA$
$typ2 : count \in \mathbb{N}$
$inv1 : reading \neq \varnothing \Rightarrow count \leq 2$
$inv2 : reading \neq \varnothing \wedge writing \neq \varnothing \Rightarrow count \leq 1$
$inv3 : reading \neq \varnothing \wedge count = 0 \Rightarrow latest \in reading$
$inv4 : wn \neq 1 \Rightarrow (\forall x \cdot x \in 0 \mathinner{..} 1 \Rightarrow wv(mem(x)) = buf(x))$
$inv5 : wn = 1 \Rightarrow buf = \{0 \mapsto d1, 1 \mapsto d1\}$

Invariant Il faut que l'écrivain se soit activé deux fois au plus (depuis le début de la lecture) si une lecture est en cours $inv1 : reading \neq \varnothing \Rightarrow count \leq 2$. Et si en plus l'écrivain est actif alors c'est une seule fois au maximum $inv2 : reading \neq \varnothing \wedge writing \neq \varnothing \Rightarrow count \leq 1$. On sait grâce à l'invariant $inv3 : reading \neq \varnothing \wedge count = 0 \Rightarrow latest \in reading$ que si une lecture est en cours et que l'écrivain ne sait jamais activé depuis alors on est toujours en train de lire l'emplacement mémoire le plus à jour. Sauf pour la première écriture, le buffer contient les valeurs écrites qui correspondent aux indices stockés dans la mémoire : $inv4 : wn \neq 1 \Rightarrow (\forall x \cdot x \in 0 \mathinner{..} 1 \Rightarrow wv(mem(x)) = buf(x))$. Et tant qu'on a fait qu'une seule écriture le buffer contient la valeur $d1$ dans ses deux emplacements : $inv5 : wn = 1 \Rightarrow buf = \{0 \mapsto d1, 1 \mapsto d1\}$.

Tous les évènements raffinent leur évènement abstrait du même nom.

Initialisation

begin

$act1 : rr := d1$

$act2 : latest := 1$

$act3 : reading := \varnothing$

$act4 : writing := \varnothing$

$act5 : buf := \{0 \mapsto d1, 1 \mapsto d1\}$

$act6 : count := 0$

end

Initialisation Les actions $act1 : rr := d1$, $act2 : latest := 1$, $act3 : reading := \varnothing$ et $act4 : writing := \varnothing$ sont les mêmes que dans l'abstraction. Nous trouvons en plus l'initialisation du buffer à la valeur $d1$ avec l'action $act5 : buf := \{0 \mapsto d1, 1 \mapsto d1\}$ et l'initialisation du compteur à zéro avec $act6 : count := 0$.

Event begin_read $\widehat{=}$

Refines begin_read

when

$grd1 : reading = \varnothing$

then

$act1 : reading := \{latest\}$

$act2 : count := 0$

end

Event end_read $\widehat{=}$

Refines end_read

any

x

where

$grd1 : x \in reading$

with

$ri : ri = mem(x)$

then

$act1 : reading := \varnothing$

$act2 : rr := buf(x)$

end

Évènement *begin_read* Il s'agit du même évènement avec en plus l'action $act2 : count := 0$ qui réinitialise le compteur puisque c'est le début d'une opération de lecture.

Évènement *end_read* L'action 2 qui réalise la lecture s'écrit maintenant $act2 : rr := buf(x)$. La variable ri n'est plus utile et disparait, pour montrer le raffinement nous avons besoin de la formule témoin $ri : ri = mem(x)$.

Event begin_write $\widehat{=}$

Refines begin_write

when

$grd1 : writing = \varnothing$

$grd2 : reading \neq \varnothing \Rightarrow count = 0$

then

$act1 : writing := \{1 - latest\}$

$act2 : count := count + 1$

end

Event end_write $\widehat{=}$

Refines end_write

any

d

x

where

$grd1 : d \in DATA$

$grd2 : x \in writing$

then

$act1 : latest := x$

$act2 : writing := \varnothing$

$act3 : buf(x) := d$

$act4 : count := count + 1$

end

Évènement *begin_write* L'action $act2 : count := count + 1$ est rajouté pour compter que l'écrivain s'est activé une fois. Et nous pouvons remplacer la deuxième garde par $grd2 : reading \neq \varnothing \Rightarrow count = 0$ à la place de $grd2 : reading \neq \varnothing \Rightarrow latest \in reading$. Cette nouvelle garde signifie qu'en cours de lecture on ne peut activer cet évènement qu'à la condition qu'il n'y ait eu aucune opération d'écriture déjà effectuée. Durant la vérification du modèle on doit prouver, à l'aide de l'invariant, le raffinement de cette garde.

Évènement *end_write* Les variables et la garde de cet évènement sont les mêmes ainsi que les actions $act1 : latest := x$ et $act2 : writing := \varnothing$. Les anciennes actions $act3$, $act4$ et $act5$ sont remplacées par $act3 : buf(x) := d$ qui écrit la nouvelle valeur dans le buffer. De même que dans l'évènement précédent on ajoute une action $act4 : count := count+1$ pour incrémenter le compteur.

4.6 Cinquième modèle : contraintes temps-réel

Dans ce raffinement, nous allons utiliser des propriétés temps-réel pour obtenir la version finale de l'algorithme. En effet pour assurer les propriétés des sûretés (exprimées dans le modèle $m3$) le lecteur et l'écrivain doivent respecter des contraintes temps-réel. Ces contraintes limitent l'asynchronisme, mais le découplage entre le lecteur et l'écrivain peut quand même être intéressant surtout que l'empreinte mémoire nécessaire est plus faible que pour la version de l'algorithme complètement asynchrone.

CONSTANTS

$minbw$ Min Between Write

AXIOMS

$axm1 : minbw \in \mathbb{N}$

$axm2 : minbw \geq 2$

Constante Soit $minbw$ (Min Between Write) définit dans un contexte associé à ce modèle avec comme axiome $axm1 : minbw \in \mathbb{N}$ et $axm2 : minbw \geq 2$. Cette constante va servir en tant que durée maximum d'une opération de lecture et en tant que durée minimum entre les opérations d'écriture. La borne inférieure de 2 sur $minbw$ est nécessaire car la durée d'une opération de lecture sera strictement inférieur à la durée entre les écritures, et nous allons considérer une durée de lecture au moins égale à une unité de temps.

VARIABLES

rr

$latest$

$reading$

$writing$

buf

cr Count Read

cnr Count Non Read

cw Count Write

cnw Count Non Write

INVARIANTS

$typ1 : cr \in \mathbb{N}$

$typ2 : cnr \in \mathbb{N}$

$typ3 : cw \in \mathbb{N}$

$typ4 : cnw \in \mathbb{N}$

$inv1 : cr < minbw$

$inv2 : writing \neq \varnothing \Rightarrow cnw \geq minbw$

$inv3 : cnw \geq minbw \wedge reading \neq \varnothing \wedge writing = \varnothing \Rightarrow count = 0$

$inv4 : writing = \varnothing \wedge reading \neq \varnothing \wedge cnw > cr \Rightarrow count = 0$

Variables et typage La variable $count$ disparait, elle représentait abstraitement les contraintes que doivent vérifier le lecteur et l'écrivain, au lieu de cela l'implémentation utilise des contraintes de temps.

Nous allons raffiner la variable $count$ par quatre variables qui sont des compteurs de durée d'un prédicat. Pour cela nous utilisons notre patron. La variable cr (Count Read) comptera la durée du prédicat $reading \neq \varnothing$, c'est-à-dire «il y a une lecture en cours», tandis que cnr (Count Non Read) comptera $reading = \varnothing$, c'est-à-dire «il n'y a pas de lecture en cours». De même pour les opération d'écriture, la variable cw (Count Write) comptera la durée du prédicat $writing \neq \varnothing$, c'est-à-dire «il y a une écriture en cours», tandis que cnw (Count Non Write) comptera $writing = \varnothing$, c'est-à-dire «il n'y a une écriture en cours». Toutes ces variables appartiennent à $\mathbb{N}$.

Invariant Une des contraintes temps-réel à respecter est de limiter la durée de la lecture, nous posons donc l'invariant $inv1 : cr < minbw$. Une autre contrainte importante est que quand le système est en train d'écrire, l'écrivain a passé au moins $minbw$ unité de temps à ne pas écrire, ce qui correspond à l'invariant $inv2 : writing \neq \varnothing \Rightarrow cnw \geq minbw$. Comme la durée de la lecture est inférieure à $minbw$, si une lecture est en cours mais pas une écriture et que cela fait au moins $minbw$ à ne pas écrire alors il n'y a eu aucune exécution de l'écrivain pendant cette lecture. Ce qui se formalise par l'invariant $inv3 : cnw \geq minbw \wedge reading \neq \varnothing \wedge writing = \varnothing \Rightarrow count = 0$. De même il n'y a pas eu d'exécution de l'écrivain pendant une lecture en cours, si le temps passé à ne pas lire est strictement supérieur à celui de la lecture : $inv4 : writing = \varnothing \wedge reading \neq \varnothing \wedge cnw > cr \Rightarrow count = 0$.

La figure 4.2 page 30 illustre les valeurs des 4 compteurs de durée. La durée comptabilisée par chaque compteur de durée (cw, cnw, cr et cnr) est illustrée par une accolade. Toutes les durées ne sont pas annotées, mais seulement une à titre d'exemple, pour chaque compteur.

En pointillé dans la colonne r on retrouve la limite que ne doit pas dépasser l'opération de lecture, sous peine d'effectuer deux opérations au même endroit de la mémoire. Un raisonnement au pire des cas permet de retrouver les contraintes temps-réel utilisées ici. Le pire moment pour commencer une lecture est juste avant la fin d'une écriture, prenons par exemple la deuxième lecture, celle sur l'emplacement 0. Dans ce cas là, la durée de lecture (cr) ne peut pas être plus longue que la durée entre deux écritures (cnw). C'est la contrainte qui est exprimée dans le modèle par l'intermédiaire de la constante $minbw$. Notons que la dernière lecture illustre le meilleur des cas, où la durée de la lecture est supérieure à $minbw$, ce cas n'est donc pas autorisé par le modèle.

Initialisation
begin
$act1 : rr := d1$
$act2 : latest := 1$
$act3 : reading := \varnothing$
$act4 : writing := \varnothing$
$act6 : buf := \{0 \mapsto d1, 1 \mapsto d1\}$
$act7 : cr := 0$
$act8 : cw := 0$
$act9 : cnr := 0$
$act10 : cnw := 0$
end

Initialisation Nous retrouvons les mêmes initialisations pour les variables déjà existantes et tous les compteurs de durée sont initialisé à zéro.

Évènements Nous allons faire en sorte que la durée de chaque opérations, c'est a dire la durée entre le début et la fin d'une opération, soit d'au moins une unité de temps. De même pour la durée de repos entre les opérations, c'est-à-dire la durée entre la fin et le début d'un opération. La raison est qu'il n'est pas réaliste de considérer des opérations instantanées. De plus cela n'a pas de sens d'arrêter et de reprendre une opération exactement au même instant. La partie du modèle qui code les quatre compteurs de durée provient tel quel du patron de raffinement des compteurs de durée.

Event begin_read $\widehat{=}$
Refines begin_read
when
$grd1 : reading = \varnothing$
$grd2 : cnr \neq 0$
then
$act1 : reading := \{latest\}$
$act2 : cr := 0$
end
Event end_read $\widehat{=}$
Refines end_read
any
x
where
$grd1 : x \in reading$
$grd2 : cr \neq 0$
then
$act1 : reading := \varnothing$
$act2 : rr := buf(x)$
$act3 : cnr := 0$
end

Évènement *begin_read* Nous ajoutons la garde $grd2 : cnr \geq 1$ pour assurer une durée non-nulle entre les opérations de lecture. Et, suivant le patron de raffinement des compteurs de durée, nous ajoutons l'action $act2 : cr := 0$ car le prédicat $reading \neq \varnothing$ devient vrai.

Évènement *end_read* De même, mais pour les durées cr et cnr, nous ajoutons la garde $grd2 : cr \neq 0$ et l'action $act3 : cnr := 0$.

Event begin_write $\widehat{=}$

Refines begin_write

when

$grd1 : writing = \varnothing$

$grd2 : cnw \geq minbw$

then

$act1 : writing := \{1 - latest\}$

$act2 : cw := 0$

end

Event end_write $\widehat{=}$

Refines end_write

any

d

x

where

$grd1 : d \in DATA$

$grd2 : x \in writing$

$grd3 : cw \neq 0$

then

$act1 : latest := x$

$act2 : writing := \varnothing$

$act3 : buf(x) := d$

$act4 : cnw := 0$

end

Évènement *begin_write* Concernant le début de l'écriture, les contraintes de l'algorithme impose que la durée entre deux écritures consécutives soit au moins de $minbw$. Nous ajoutons donc la garde $grd2 : cnw \geq minbw$. Et conformément au patron on ajoute l'action $act2 : cw := 0$.

Évènement *end_write* L'évènement de la fin de l'écriture est standard avec la garde $grd3 : cw \neq 0$ et l'action $act4 : cnw := 0$.

Event tic $\widehat{=}$

any

ncr

$ncnr$

ncw

$ncnw$

where

$grd1 : reading \neq \varnothing \Rightarrow cr \leq minbw - 2$

$grd2 : ncr \in \mathbb{N} \wedge ncnr \in \mathbb{N} \wedge ncw \in \mathbb{N} \wedge ncnw \in \mathbb{N}$

$grd3 : reading \neq \varnothing \Rightarrow ncr = cr + 1 \wedge ncnr = cnr$

$grd4 : reading = \varnothing \Rightarrow ncr = cr \wedge ncnr = cnr + 1$

$grd5 : writing \neq \varnothing \Rightarrow ncw = cw + 1 \wedge ncnw = cnw$

$grd6 : writing = \varnothing \Rightarrow ncw = cw \wedge ncnw = cnw + 1$

then

$act1 : cr := ncr$

$act2 : cnr := ncnr$

$act3 : cw := ncw$

$act4 : cnw := ncnw$

end

Évènement *tic* Conformement au patron, on incrémente les quatre durées cr, cnr, cw et cnw pour cela nous utilisons les variables locales ncr, $ncnr$, ncw et $ncnw$ qui sont les nouvelles valeurs des durées. Si le prédicat associé à la durée, qui se résume ici à une expression booléenne, est vraie la nouvelle valeur de la durée est la vieille plus 1. On obtient ceci à l'aide des gardes :

$grd2 : ncr \in \mathbb{N} \wedge ncnr \in \mathbb{N} \wedge ncw \in \mathbb{N} \wedge ncnw \in \mathbb{N}$

$grd3 : reading \neq \varnothing \Rightarrow ncr = cr + 1 \wedge ncnr = cnr$

$grd4 : reading = \varnothing \Rightarrow ncr = cr \wedge ncnr = cnr + 1$

$grd5 : writing \neq \varnothing \Rightarrow ncw = cw + 1 \wedge ncnw = cnw$

$grd6 : writing = \varnothing \Rightarrow ncw = cw \wedge ncnw = cnw + 1$

Jusqu'ici tout est conforme au patron, mais nous devons ajouter, en plus, la garde $grd1 : reading \neq \varnothing \Rightarrow cr \leq minbw - 2$ pour exprimer la contrainte temps-réel de limiter la durée de l'opération de lecture. Au plus, cette durée est de $minbw - 1$ et donc avant l'exécution de *tic*, elle est de $minbw - 2$. Ainsi il n'y a pas d'instant pendant lequel on arrête d'écrire et où on commence à lire en même temps, comportement qui peut être considéré comme erroné.

Action Les action ne consistent qu'à substituer les nouvelles valeurs déterminées dans la garde avec : $act1 : cr := ncr$, $act2 : cnr := ncnr$, $act3 : cw := ncw$ et $act4 : cnw := ncnw$.

Non-blocage Dans la méthode B évènementielle, il existe une obligation de preuve, dite de non-blocage, pour montrer qu'il n'existe pas un état non désiré dans lequel aucun évènement n'est capable de se déclencher. Pour cela il faut démontrer la disjonction des gardes des évènements étudiés. Notre algorithme décrit un système réactif perpétuel, nous avons donc vérifié le théorème $thm1$, la disjonction des gardes de tous les évènements :

$$
\begin{array}{l}
thm1 : (reading = \varnothing \wedge cnr \neq 0) \vee \\
\quad (reading \neq \varnothing \wedge cr \neq 0) \vee \\
\quad (writing = \varnothing \wedge cnw \geq minbw) \vee \\
\quad (writing \neq \varnothing \wedge cw \neq 0) \vee \\
\quad \left(\begin{array}{l}
\exists ncr, ncnr, ncw, ncnw \cdot \\
\quad ncr \in \mathbb{N} \wedge ncnr \in \mathbb{N} \wedge ncw \in \mathbb{N} \wedge ncnw \in \mathbb{N} \wedge \\
\quad (reading \neq \varnothing \Rightarrow cr \leq minbw - 2) \wedge \\
\quad (reading \neq \varnothing \Rightarrow ncr = cr + 1 \wedge ncnr = cnr) \wedge \\
\quad (reading = \varnothing \Rightarrow ncr = cr \wedge ncnr = cnr + 1) \wedge \\
\quad (writing \neq \varnothing \Rightarrow ncw = cw + 1 \wedge ncnw = cnw) \wedge \\
\quad (writing = \varnothing \Rightarrow ncw = cw \wedge ncnw = cnw + 1)
\end{array} \right)
\end{array}
$$

Il faut noter que l'évènement *tic* de progression du temps est inclus dans cette preuve de non-blocage. Il est en effet possible d'introduire des contraintes temps-réel conduisant à un blocage du système, situation qu'il faut interdire.

Enfin notons que nous avons simplifié les formules du type $\exists x \cdot x \in S$ en $S \neq \varnothing$.

4.7 Vérification

Ce développement a été conçu sur le logiciel Rodin du projet européen du même nom. Toutes les obligations de preuves ont été déchargées. Rodin utilise le prouveur B4Free de ClearSy. Le tableau suivant donne les détails du nombre d'obligations de preuves par modèles :

Modèle	Totales	Automatiques	Manuelles	Non déchargées
m0	34	25	9	0
m1	12	12	0	0
m2	42	39	3	0
m3	33	28	5	0
m4	33	30	3	0

4.8 Conclusion

Nous avons développé, par la preuve, une version d'un algorithme de communication asynchrone (l'algorithme de Simpson [9]) utilisant des contraintes temps-réel pour résoudre un problème d'accès concurent. La communication se fait depuis un processus «écrivain» vers un processus «lecteur». Pour cela le système dispose d'une mémoire partagée par les deux processus. La version étudiée ici possède un buffer à deux emplacements. L'écrivain va écrire alternativement dans les deux emplacements et le lecteur va lire le dernier emplacement mémoire modifié. Pour que cela fonctionne, il ne faut pas que l'écrivain réalise deux écritures pendant la durée d'une lecture, sinon les deux processus accèdent au même emplacement en même temps.

Nous avons donc développé, dans les premiers modèles, l'algorithme en quatre étapes en remplaçant les contraintes temps-réel par des conditions plus abstraites d'ordonnancement, tel que la condition décrite ci-dessus. Nous avons ensuite appliqué notre patron de compteur de durée dans le dernier modèle, ce qui a permis de remplacer les conditions abstraites d'ordonnancement par des conditions sur les durées d'exécution des deux processus. Nous avons ainsi expérimenté, avec succes, ce patron et montré que les contraintes temps-réel implémentent bien la spécification donnée. Cette étude de cas montre une application du patron et montre la réutilisation d'une solution générique à un problème temps-réel de modélisation et de preuve.

Chapitre 5

Conclusion

Au cours de ce livrable, nous avons proposé un concept d'agenda et un patron pour modéliser les contraintes temps-réel dans la méthode B évènementielle. Ce patron gère un ensemble d'actions dites temps-réel auquelles nous associons à chacun un ensemble de futures occurrences d'exécution. Ces temps sont ajoutés dans les modèles quand l'état courant du système permet de les prédire, ils sont ensuite utilisés pour contraindre le déroulement du système dans le temps. Ce concept est inspiré par [7] et [5]. Nous expliquons comment utiliser ce patron de modèle du temps en raffinant les évènement du patron vers les évènements du système étudié. Des exemples sont donné.

Nous proposons aussi un concept temps-réel appellé compteur de durée de prédicat. Il s'agit de connaitre, pour un système dynamique, la durée depuis laquelle un prédicat est vrai ou la durée pendant laquelle ce prédicat a été vrai. Ce concept est inspiré par [1] et [4]. De même que pour l'agenda, nous proposons une définition et un modèle du concept sous la forme d'un patron de raffinement à appliquer.

Le document poursuit par une étude de cas menée sur une partie du standard IEEE de la norme FireWire. Il s'agit de l'élection entre deux périphériques FireWire communiquant de manière asynchrone. Cette expérimentation montre l'application du patron sur un cas industriel, en particulier nous montrons que la démarche incrémentale propre à la méthode B est bien permise par notre patron. Le développement des modèles avec leurs invariants et les relations de raffinement sont complétement formalisés et sont prouvés à l'aide de B4Free et Click'n'Prove.

Une deuxième étude de cas est fournie en tant qu'expérimentation du patron des durées. Il s'agit d'un algorithme de communication asynchrone nécessitant des contraintes temps-réel pour assurer la sureté de son fonctionnement. Le développement des modèles est réalisé et prouvé avec Rodin et B4Free.

Des travaux sont en cours pour adapter notre démarche à une vérification par model-checking. Il s'agit d'utiliser une version de l'agenda avec des valeurs relatives qui sont plus adaptées à la nature de la vérification par model-checking. Nous nous proposons également de poursuivre l'étude de cas de l'algorithme de Simpson, cet algorithme peut servir de brique de base au sein de systèmes temps-réel plus complexes nécessitant des communications asynchrones.

Bibliographie

[1] Martín Abadi and Leslie Lamport. An old-fashined recipe for real-time. *ACM Trans. Program. Lang. Syst.*, 16(5) :1543–1571, 1994.

[2] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A mechanically proved and incremental development of IEEE 1394 tree identify protocol. *Formal Asp. Comput.*, 14(3) :215–227, 2003.

[3] Jean-Raymond Abrial, Dominique Cansell, and Dominique Méry. A new IEEE 1394 leader election protocol. In *Rigorous Methods for Software Construction and Analysis Seminar N 06191,07.05.-12.05.06*. Schloss Dagstuhl, U. Glaser and J. Abrial, 2006.

[4] Rajeev Alur and David L. Dill. A theory of timed automata. *Theor. Comput. Sci.*, 126(2) :183–235, 1994.

[5] Rajeev Alur, Limor Fix, and Thomas A. Henzinger. Event-clock automata : A determinizable class of timed automata. *Theor. Comput. Sci.*, 211(1-2) :253–273, 1999.

[6] Dominique Cansell, Dominique Méry, and Joris Rehm. Time constraint patterns for event B development. In Jacques Julliand and Olga Kouchnarenko, editors, *B*, volume 4355 of *Lecture Notes in Computer Science*, pages 140–154. Springer, 2007.

[7] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In Yassine Lakhnech and Sergio Yovine, editors, *FORMATS/FTRTFT*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2004.

[8] Joris Rehm and Dominique Cansell. Proved Development of the Real-Time Properties of the IEEE 1394 Root Contention Protocol with the Event B Method. In Frederic Boniol Yamine Aït Ameur and Virginie Wiels, editors, *RNTI ISoLA 2007 Workshop On Leveraging Applications of Formal Methods, Verification and Validation*, volume RNTI-SM-1, pages 179–190, Poitiers-Futuroscope France, 12 2007. Cépaduès.

[9] H.R. Simpson. Four-slot fully asynchronous communication mechanism. *Computers and Digital Techniques, IEE Proceedings* -, 137(1) :17–30, Jan 1990.

[10] Mariëlle Stoelinga. Fun with firewire : A comparative study of formal verification methods applied to the ieee 1394 root contention protocol. *Formal Asp. Comput.*, 14(3) :328–337, 2003.